NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

CALCULATION OF TARGET MASKING EFFECTS FOR AIR TO SURFACE ORDNANCE DELIVERY

by

Joseph R. Darlak

December, 1996

Thesis Advisor: Morris Driels

Approved for public release; distribution is unlimited.

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1996	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE CALCULATION OF TARGET MASKING EFFECTS FOR AIR TO SURFACE ORDNANCE DELIVERY

5. FUNDING NUMBERS

6. AUTHOR(S) Joseph R. Darlak

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Naval Postgraduate School
Monterey CA 93943-5000

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

The Perspective View Generator (PVG) is a database driven simulation that displays a three-dimensional synthetic environment from terrain information stored in a two-dimensional array. Each terrain element is identified by coordinates which index the array. Each array value contains 32 bits of information arranged to specify the greyshade, elevation and other features of the terrain at that position. Currently there are few areas of the world digitally mapped in the data format accepted by the PVG.

The objective of this thesis is to enable the automatic generation of a database for use with the PVG from information available from various sources. The project involves writing computer algorithms to synthesize data from different sources and making modifications to the PVG functions. The source of terrain information is aerial imagery and elevation information which is not resolved fine enough to distinguish tree and building heights. The ability to modify the data base to include object heights allows the PVG to show the target masking effects these objects have in a given region. The resulting synthetic environment can be used for strike planning and as a mission tailored training tool.

14. SUBJECT TERMS Synthetic environments, terrain, visualization, target masking			15. NUMBER OF PAGES 89 16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18 298-102

Approved for public release; distribution is unlimited.

CALCULATION OF TARGET MASKING EFFECTS FOR AIR TO SURFACE ORDNANCE DELIVERY

Joseph R. Darlak
Lieutenant, United States Navy
B.S.M.E., United States Naval Academy, 1990

Submitted in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN MECHANICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
December 1996

ABSTRACT

The Perspective View Generator (PVG) is a database driven simulation that displays a three-dimensional synthetic environment from terrain information stored in a two-dimensional array. Each terrain element is identified by coordinates which index the array. Each array value contains 32 bits of information arranged to specify the greyshade, elevation and other features of the terrain at that position. Currently there are few areas of the world digitally mapped in the data format accepted by the PVG.

The objective of this thesis is to enable the automatic generation of a database for use with the PVG from information available from various sources. The project involves writing computer algorithms to synthesize data from different sources and making modifications to the PVG functions. The source of terrain information is aerial imagery and elevation information which is not resolved fine enough to distinguish tree and building heights. The ability to modify the data base to include object heights allows the PVG to show the target masking effects these objects have in a given region. The resulting synthetic environment can be used for strike planning and as a mission tailored training tool.

TABLE OF CONTENTS

ACKNOWLEDGEMENT

The author would like to acknowledge USAMSAA for sponsoring this project.

The author wants to thank Prof. Driels for his guidance, Mr Dale Robison from NAWC, China Lake, for providing the computer program SELECT and the DTED and SPOT data and finally to extend very special thanks to Christine and Gabriella for their support, patience and love.

I. INTRODUCTION

The ability to conduct realistic training is a crucial component of military readiness. Sailors, ground troops and pilots all benefit from opportunities to hone their craft. Due to decreasing Defense and training budgets, commands have become expert at incorporating training drills into normal evolutions. This opportunistic training is excellent however, there are many drill scenarios that can not be exercised due to operational commitments or due to their cost. Whenever training is conducted on operating gear, there is always a chance that mistakes can harm personnel or equipment. As Defense dollars dwindle and the size of the military shrinks, the list of evolutions that are too expensive will only get longer.

There is no substitute for "live" training, however, realistic simulators can provide a cost effective bridge from the raw recruit to the professional warrior. Propulsion, combat systems, firefighting and flight simulators are just a few examples of the many training simulations employed by the United States Armed Forces. These simulators provide realistic experience in a controlled environment where personnel and equipment casualties are limited. They also provide the opportunity to obtain a high level of proficiency through repetition and at minimal interference with operational commitments. The high level of competence achieved in the simulators pays dividends of readiness and reduced personnel and equipment casualties in the real world.

Synthetic environments or virtual reality scenarios offer tremendous potential for the conduct of training and planning. If optimally utilized they can save lives, equipment and

money. The goal of synthetic environments is to create the perception of normal movement through a realistic scene. To achieve this goal, a program must display detailed scenes at a rate not less than 15 image frames per second. [Ref. 1: p. 3]

A. OVERVIEW OF SYNTHETIC ENVIRONMENTS

Synthetic environments are divided into two general categories: object based simulations and database driven simulations. Object based simulations insert objects into an initially blank scene. The simulation then performs calculations to construct these objects to conform to the user's field of view. Any movement in the simulation requires a reconstruction of each object to determine its masking effect on other objects due to the change in viewpoint. This method works well for scenes with small numbers of simple objects. Ultimately the desire for realism requires that more objects of increasing detail and complexity be included. The individual calculations required for each object become more computationally expensive as the level of detail is increased. These numerous and time consuming calculations rapidly decrease the speed of object based simulations when detailed scenes are attempted. In order to maintain an adequate frame rate the realism of the objects in the scene must be sacrificed.

In a database driven simulation, all the information about a given section of terrain and the objects on it is stored in a database. The calculations required to define all the objects are performed once and the resulting database will determine what is displayed to the user. A large number of calculations are still required to determine what the user should see as he/she moves about, but this number is relatively constant regardless of the complexity

of the scene. It is this feature of the database driven simulation which allows movement in very complex scenes at frame rates large enough to achieve a high degree of realism. [Ref. 1: p. 3]

B. THE PERSPECTIVE VIEW GENERATOR (PVG)

The PVG is a database driven synthetic environment simulation. It consists of a collection of C programs written to generate a three-dimensional view using the graphic capabilities of a Silicon Graphics workstation. The PVG user controls his/her movement over the terrain with a mouse or spaceball. The PVG uses the concepts of a field of view and a boresight vector to calculate which portion of the terrain the user sees. The outcome of these calculations determine which elements in the PVG database are displayed. In this section references to the original PVG are to the PVG as it existed prior to the modifications made as a result of this thesis.

The database for the PVG consists of a two-dimensional array. Each element in the array corresponds to a one square meter section of terrain. The array is indexed by the x and y coordinates of the terrain sections represented. The information encoded in each array element includes greyshade, terrain elevation above sea level, heights of objects above the terrain and other necessary characteristics which will be described later.

1. Mechanics of Ray Tracing

To imagine how the PVG works, consider the screen as being the view from a black and white video camera. When looking through it, what the observer sees depends on the field of view of the camera. A vector, called the boresight vector, can be thought of as originating from the center of the camera lens normal to its surface. The boresight vector

points in the direction that the camera is pointing and is specified in terms of x, y, and z coordinates along with two direction cosines. If the observer is oriented such that he/she is looking down at the terrain, the boresight vector may be followed down in incremental steps until it intersects the terrain. The concept of following the boresight vector to the terrain is called a ray trace. Information about the intersection point can then be retrieved from the database. This information will correspond to what is seen at the center pixel of the screen window or in the middle of the camera view. Figure 1.1 shows the boresight vector extending from the screen toward the terrain.

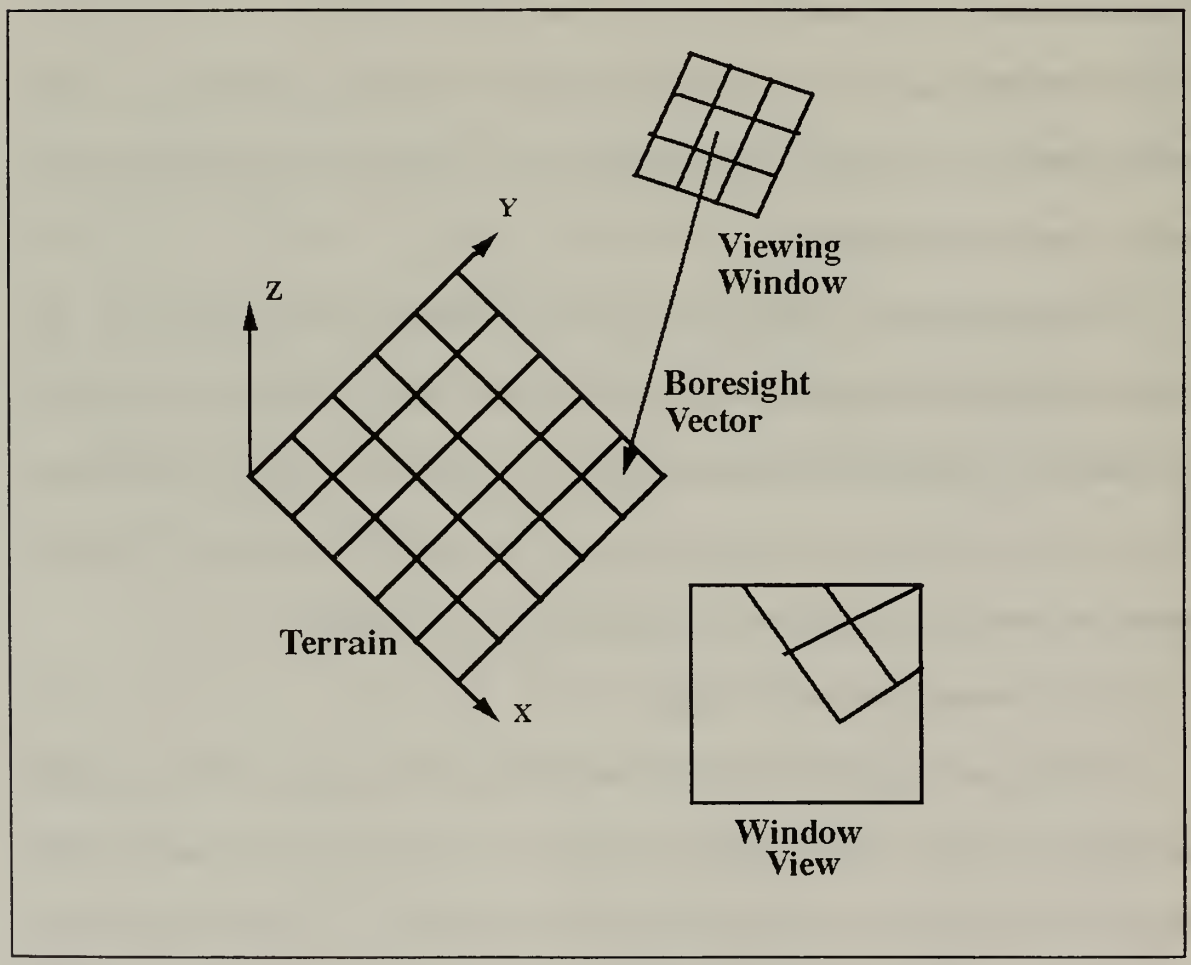

Figure 1.1. Relationship Between Terrain and Field of View. After Ref.[1].

The field of view of the camera would actually be a solid cone extending to the terrain. In the PVG, the angular field of view is distributed in equal increments of elevation and azimuth across the dimensions of the viewing window or screen. Since the viewing window is a square, the PVG's field of view extends as a pyramid. Each increment in the field of view is associated with a fixed number of screen pixels. Once the direction cosines of the boresight vector, at the center pixel, are defined it is possible to define vectors from each of the screen pixels in the field of view relative to the boresight. These vectors extend toward the terrain within the pyramid field of view. The result is that each screen pixel's vector can be traced to the terrain and the information stored in the database where the trace intersected the terrain can be displayed. Input received from the mouse or spaceball is used to change the location and orientation of the boresight vector. [Ref. 1: pp. 4-6]

2. Organization of Information in the Data Array

Each element of the data array is allotted 32 bits to encode information about the particular square meter of terrain it represents. Figure 1.2 is a summary of the bit fields. In the original PVG the first six bits assign the greyshade. The seventh bit is the sun/shade bit which provides information about the position of the sun for the calculation of shadows. In practice the utility of this information was low so all shadows in the PVG are calculated with the sun positioned directly overhead. The eighth bit is the nature bit and it distinguishes between natural and man-made objects on the terrain. The ninth and tenth bits are reserved for vegetation identification. The PVG does not use this information. Bits 11 through 14 store an object's height above the terrain in meters. The 15th through 18th bits are reserved

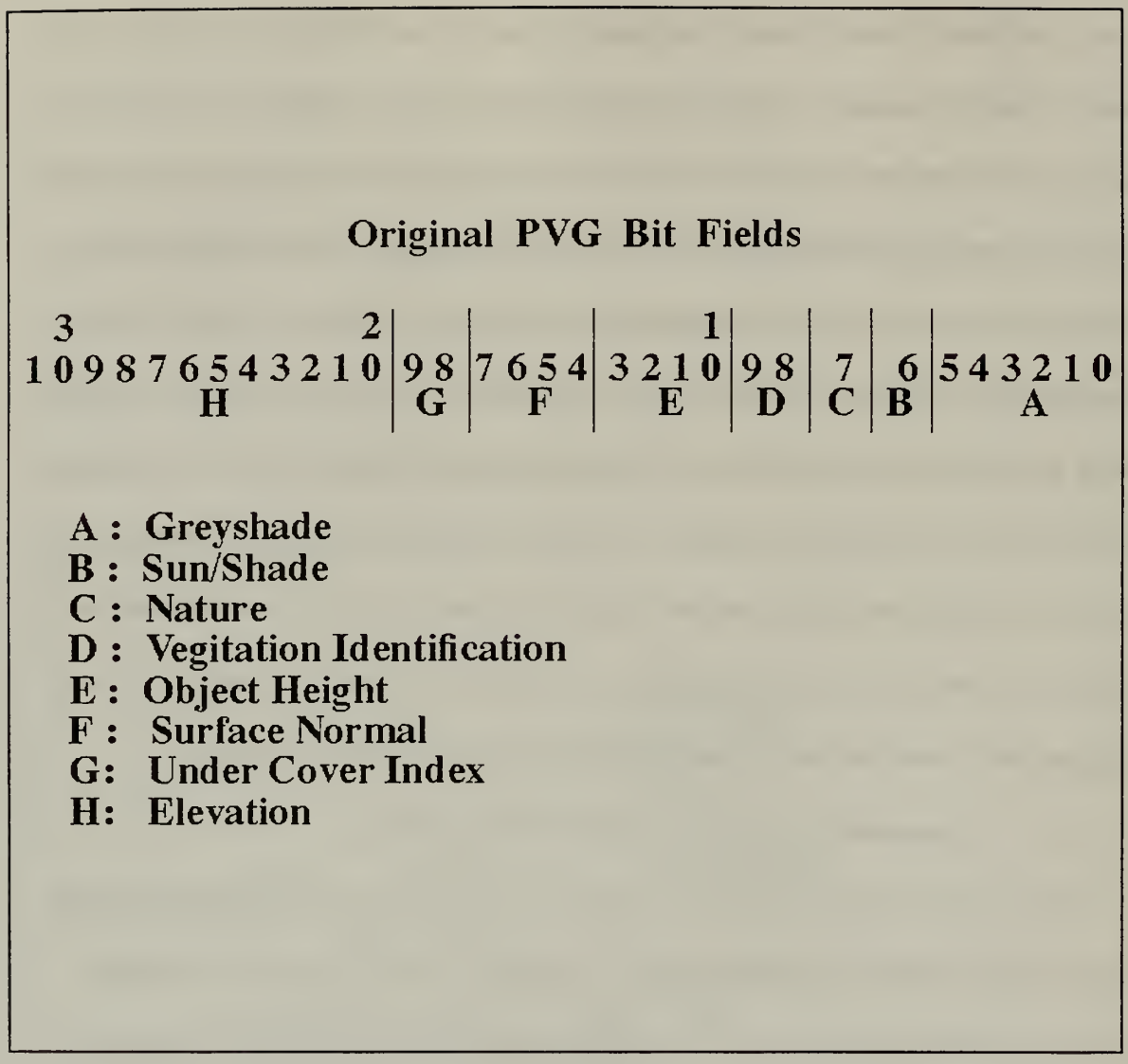

Figure 1.2. Summary of PVG Bit Fields. After Ref.[2].

for the surface normal's of objects above the terrain surface. The original PVG does not utilize the surface normal information. Bits 19 and 20 store the under cover index. The under cover index is a value in meters measured from the terrain to an object that overhangs the terrain like the leaf canopy of a tree. The remaining bits store elevation information in half meters. Elevation in the PVG is defined as the distance from sea level to the top of an object on the terrain. In the original PVG, the program *datagen.c* fills a two-dimensional

array called DAT[x][y] with the bit field values taken from a previously compiled database. Figure 1.3 shows a graphical interpretation of the bit fields.

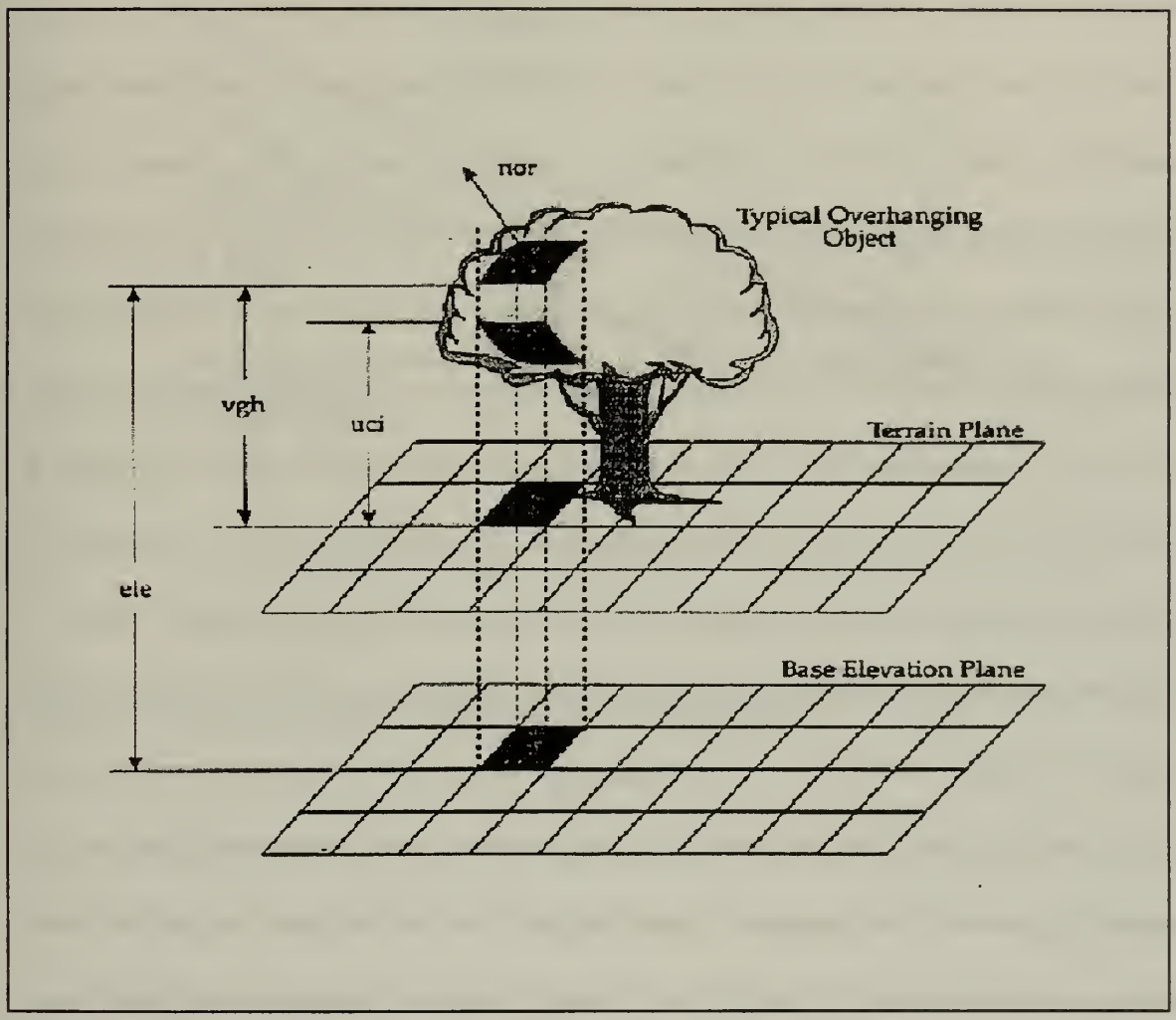

Figure 1.3. Graphical Representation of Dimensional Bit Fields. In this Figure vgh is equal to Object Height. From Ref.[2].

3. Information Retrieval

The ray trace algorithm introduced earlier is the mechanism of data retrieval and is performed in the function called *zigzag.c*. The observer is generally some distance above the terrain with the boresight vector making an angle between 0 and 90 degrees with the

plane of the terrain. The actual ray tracing starts with the lower left screen pixel. The trace proceeds in one meter steps along the ray originating from the center of the pixel. For each step that it is above the terrain, the z coordinate of the ray is compared to the elevation value stored in the database for the particular x and y coordinates the step point is over. As long as the step point remains above the elevation the trace will continue to take one meter steps toward the terrain. Once the z coordinate of the ray trace is less than or equal to the elevation, the information stored in the database for the corresponding x and y is retrieved. A similar trace is then conducted for the pixel in the next row of the same column until the entire column is completed. The column is then incremented until all the pixel rays have been traced. Analysis of the PVG demonstrated that 70 to 80% of the time to generate a screen image was spent in the ray tracing algorithm. To make the ray trace more efficient a technique known as zig-zag is applied to successive traces in the same column. [Ref. 1: p. 18] With the observer positioned as before, some distance above the terrain, the first ray trace will require n steps before encountering the terrain. The next higher pixel in the column would then require n+p steps to reach the terrain. To save computing time the ray trace for the second pixel is begun n steps along its ray so the only time required to finish its trace is the time required to take the last p steps. In this way the number of trivial steps taken is significantly reduced. Figure 1.4 shows the zig-zag method.

When the ray intersects the terrain the program *hitgrnd.c* is called to interpret the information stored in the data array for that pixel. Logical masks are used to retrieve the values of the bit fields individually. In the PVG a mask is a 32 bit value that has ones in the bit locations of the field of interest and zeros everywhere else. For example the mask for

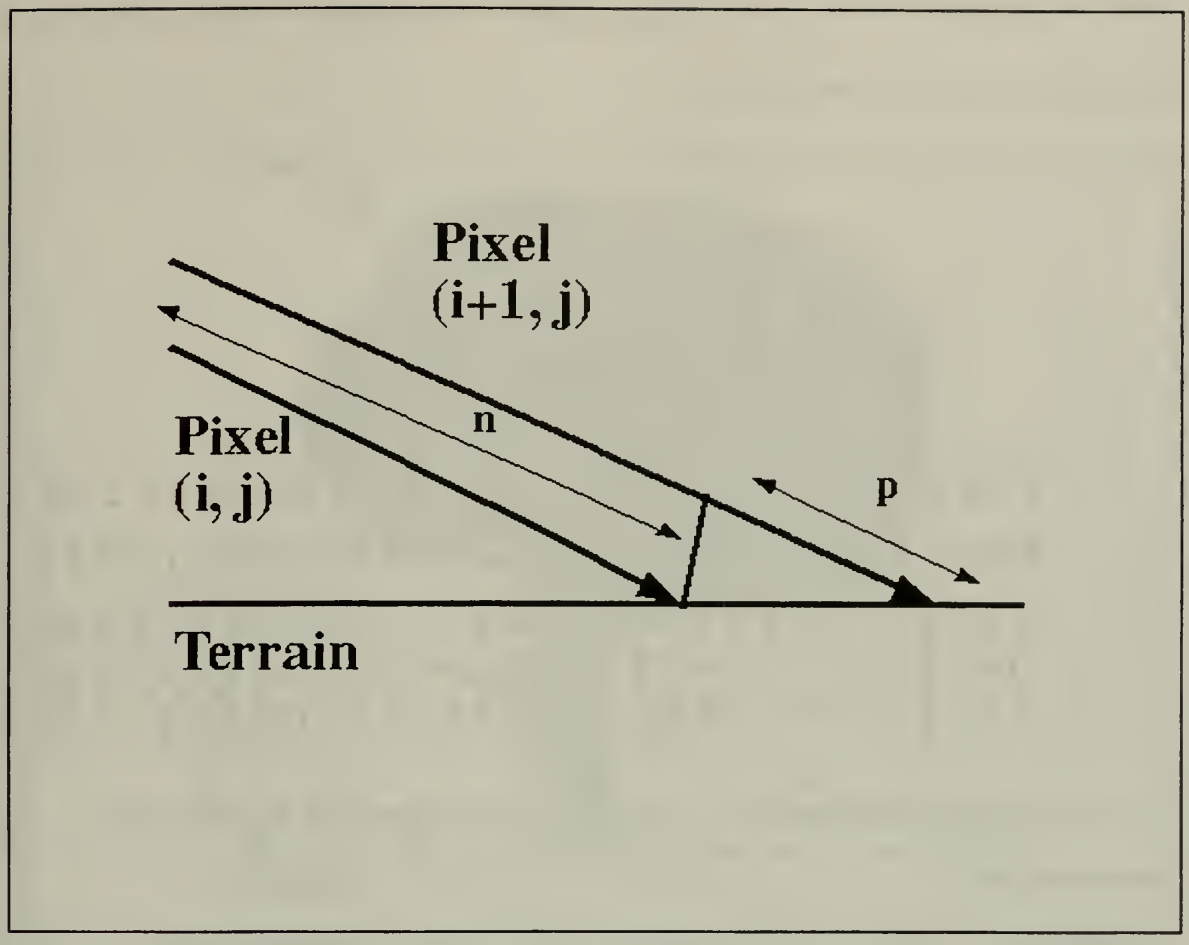

Figure 1.4. Zig-Zag Feature of Ray Trace. After Ref.[1].

object height would have ones at the 11th through 14th bit locations and zeros everywhere else. When this mask and a value from the data array are combined with a logical, bitwise AND operator the result is that the bit values at the 11th through 14th locations survive and the rest go to zero. In this way the value assigned for object height is isolated, however this value is not the actual object height above the terrain. To convert to a meaningful object height, the result of the AND operation described above must be shifted 11 bits to the right so that bit 11 becomes the least significant bit. This shifted value is the height of the object

above the ground in meters. In a similar way all the bit fields can be retrieved. Figure 1.5 summarizes the bitwise operations.

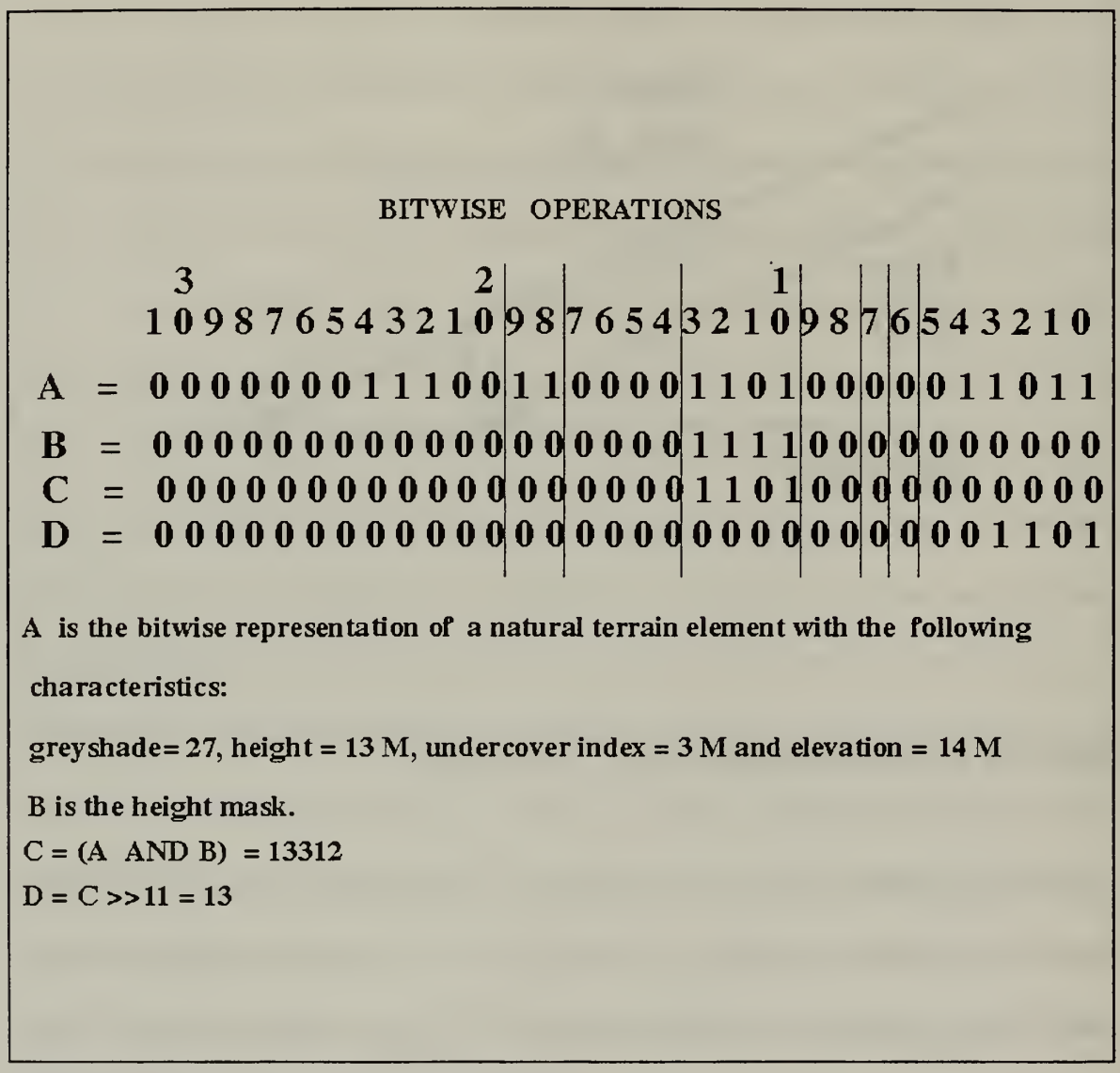

Figure 1.5. Summary of Bitwise Operations.

In the original PVG, when the ray trace strikes the terrain *hitgrnd.c* first checks if the terrain is bald, where bald terrain means terrain without objects. If the terrain is bald then the greyshade is read from the data array and it is returned to *zigzag.c* where this value is placed in the image buffer at the location corresponding to the screen pixel. In Figure 1.6,

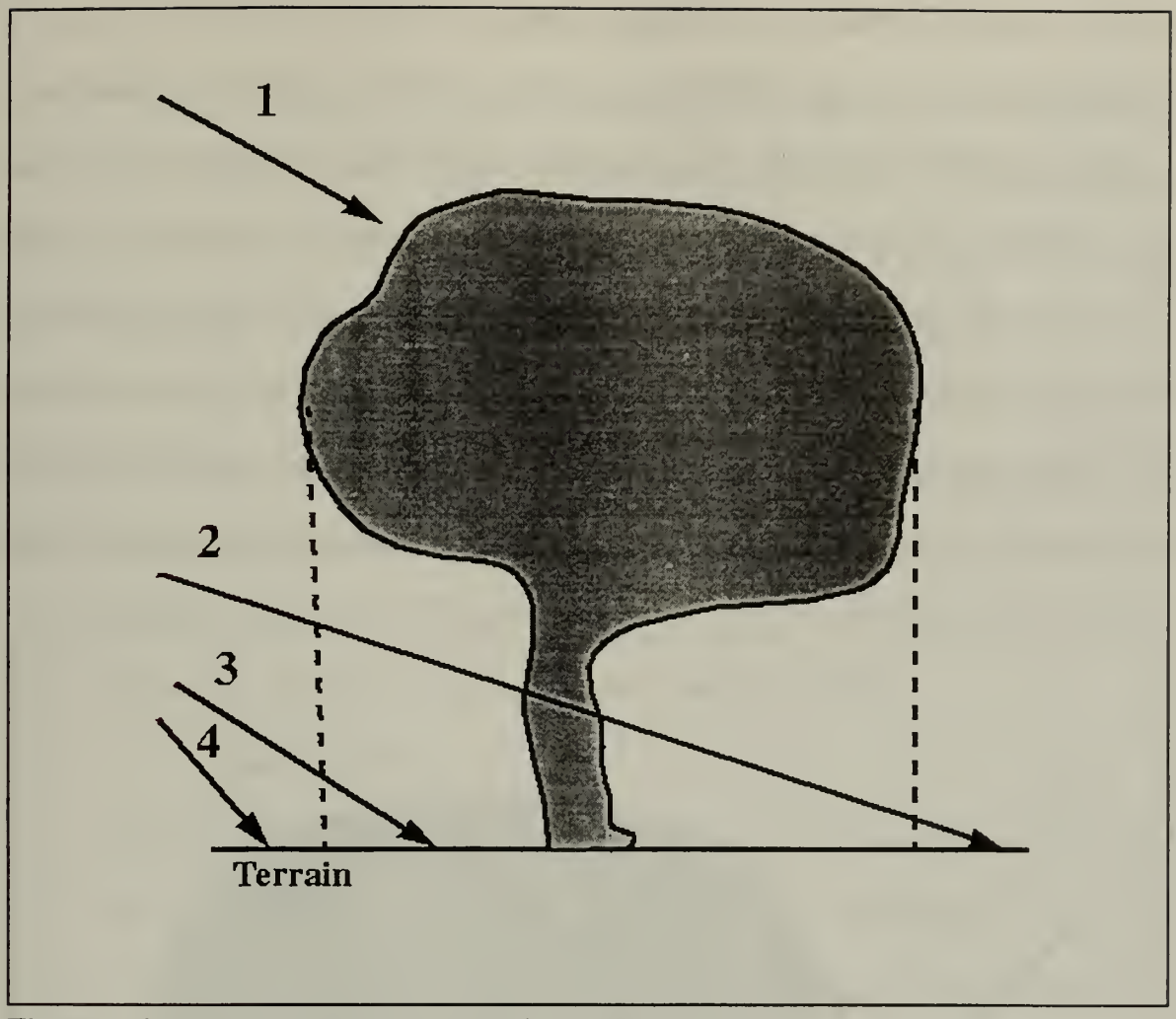

Figure 1.6. Ray Interaction with an Object. After Ref.[1].

ray number 4 strikes bald terrain. If the terrain is not bald the next check is to determine if the location is under the overhang of an object. If the location is under an overhang, black is returned to the image buffer representing a shadow in the final image. Ray number 3 would return a shadow. If the ray strikes inside the overhang then the greyshade is taken from the database and returned to the buffer. Ray number 1 strikes the canopy and the greyshade of the object, in this case a tree, is returned. The final case is a ray with a z coordinate that is between the terrain and the bottom of an overhang. In this case the ray

trace is continued from this original intersection point until either an object or the terrain is encountered and a greyshade or shadow will be placed in the image buffer. Ray number 2 passes under the tree canopy and strikes bald terrain on the other side. After all the pixels have their appropriate greyshade, the image frame is ready for viewing. [Ref. 1: pp. 16-20]

The zig-zag method offers a considerable advantage by minimizing the amount of time spent in the ray trace but there is a pitfall which must be considered. As seen in Figure 1.7, after ray number 1 strikes the terrain the zig-zag jumps up to ray number 2 at point C. The trace for ray number 2 then proceeds from point C till it intersects the tree trunk at point

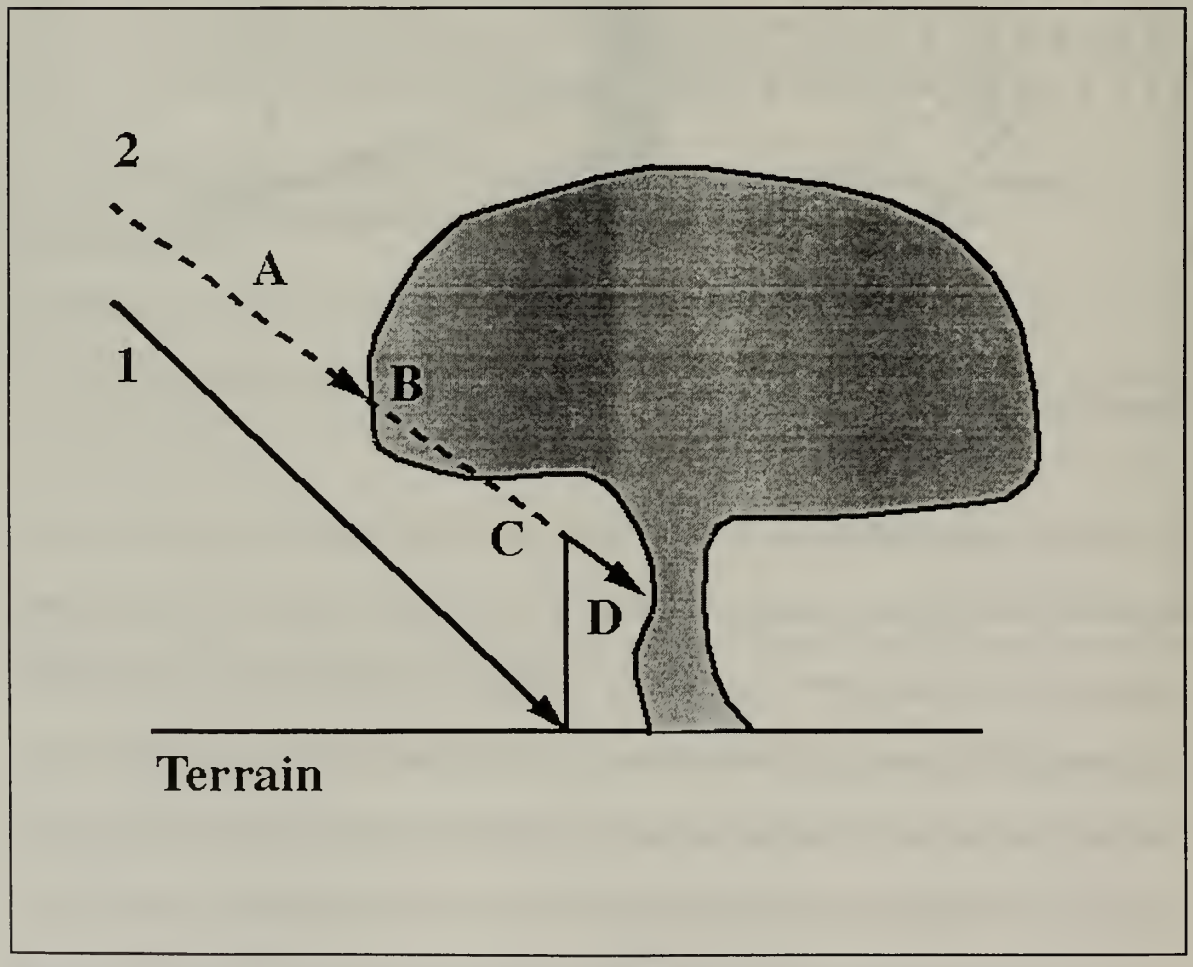

Figure 1.7. Effect of Zig-Zag on Ray Interaction with an Overhanging Object. After Ref.[1].

D and the information stored at the location of point D is sent to the image buffer. If one extends the untraced portion of ray number 2 back toward the observer however, it can be seen that ray number 2 would really have intersected the tree in the leaf canopy at point B. Without considering this pitfall images in the PVG would be distorted and unrealistic. In the case of Figure 1.7, the tree would appear as a tall trunk with little or no leaf canopy. To avoid this pitfall a back-step is used with the zig-zag. After ray number 1 strikes the terrain it still jumps up to ray number 2 but instead of starting the next trace at point C, the starting point is stepped back a fixed distance to point A. With the trace of ray number 2 now beginning at point A, the leaf canopy is intersect at point B and this information is sent to the image buffer. The leaf canopy is then clearly seen in the PVG.

4. Image Rendering

Depending on the configuration of the PVG several windows can be opened at the start of the run. These windows are opened using GL library functions. Usually one or more windows will be used to display a static scene from the viewpoint of an observer directly above the scene area looking straight down. The remaining window is used for the dynamic view. Figure 1.8 shows an example of the multiple windows displayed on the screen while running the PVG. The image on the left is a downward looking static view. The one on the right is the dynamic view and the lower window is a UNIX Shell for user interaction. To display the scene stored in the image buffer two commands are utilized. First, rectzoom is used to size the image buffer appropriately to fit in the window. The second command, lrectwrite, then writes the zoomed image buffer to the screen. This entire process is then repeated for the next boresight vector orientation.

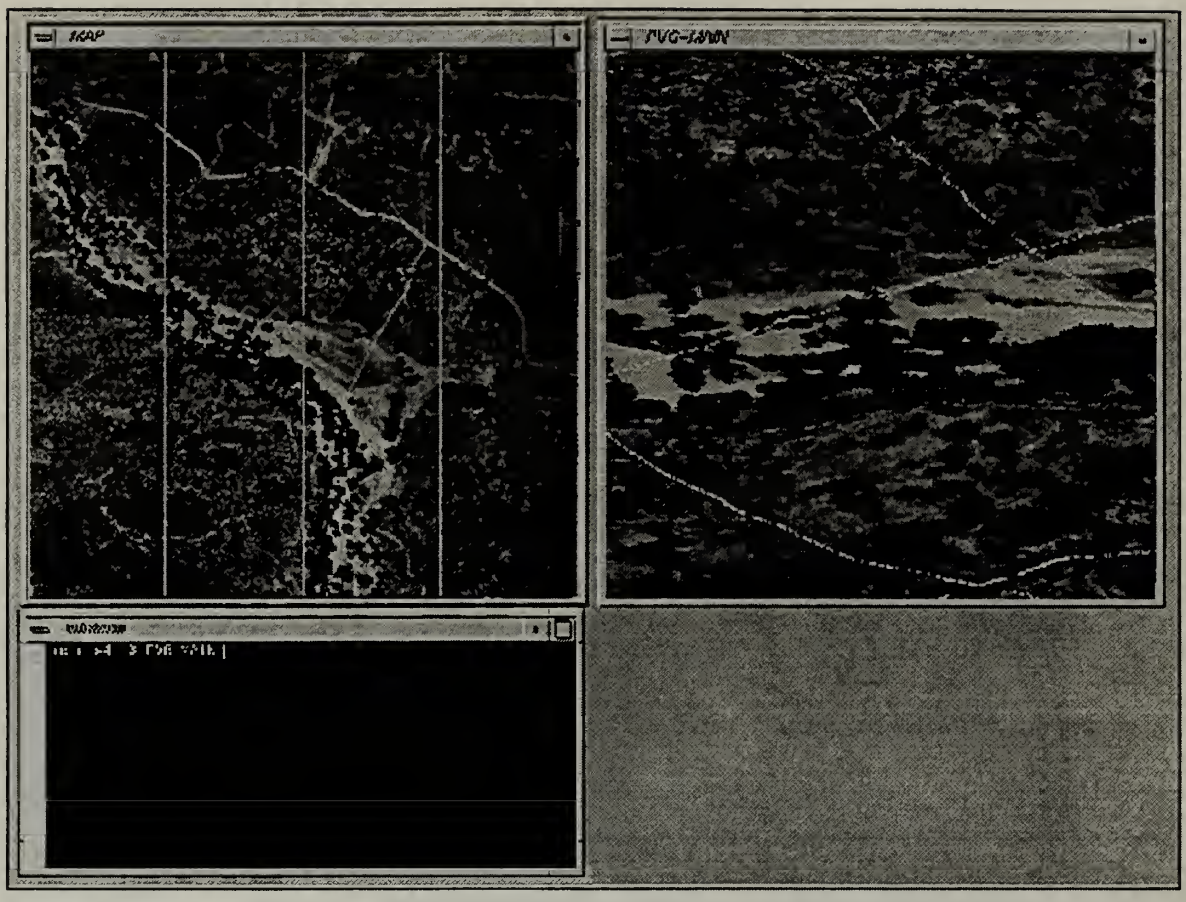

Figure 1.8. Sample PVG Screen Arrangement.

Figures 1.9 and 1.10 are enlarged static and dynamic views respectively. These images are of the United States Army Fort, Fort Hunter Liggett in central California. They are shown as displayed by the original PVG.

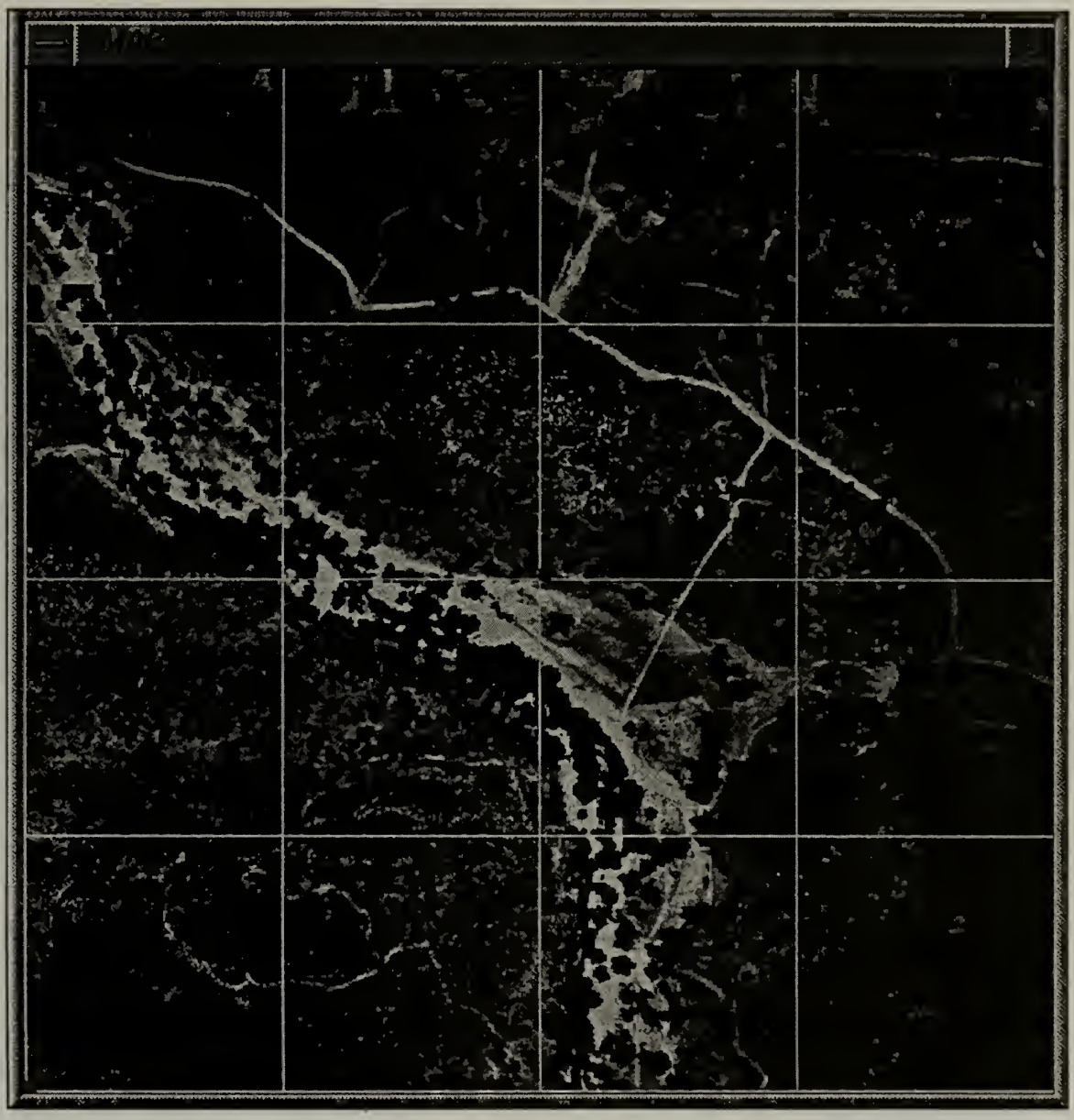

Figure 1.9. PVG Static View.

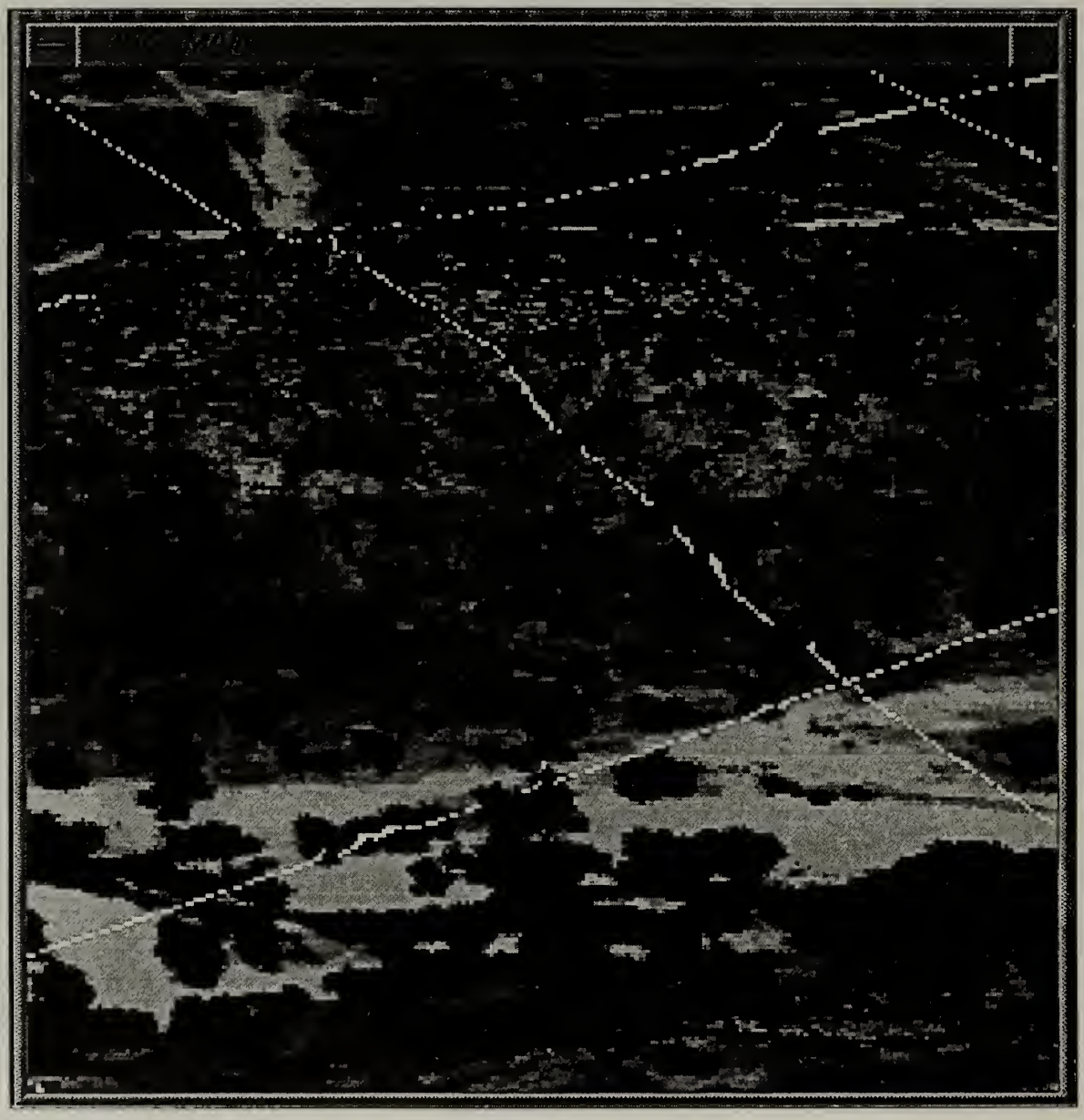

Figure 1.10. PVG Dynamic View.

II. OBJECTIVE

The database used by the original PVG contained very detailed information about Fort Hunter Liggett. The construction of this database was an extensive undertaking which required considerable effort and time The objective of this thesis is to take an image of an area and some topographical information and generate a database which will result in a realistic simulation when used with the PVG. Ideally the process will be general enough that given a minimum of required information any terrain on this or any other planet can be simulated. The primary requirements for the database are greyshade and topography. With virtually the entire Earth's surface accessible by satellite and a comprehensive amount of topographical information available, success with this objective would mean that nearly the entire Earth's surface could quickly and easily be rendered.

The motivation for this project is found in military requirements. The necessity of pin point strikes with minimal collateral damage and casualties requires carefully planned and rehearsed missions. By demonstrating the masking effects that large objects may have on possible targets located near them, the PVG can be used as a tool by commanders, mission planners and those chosen to execute the mission. The PVG can be used to determine which targets, from a list of candidates, could be most easily attacked from a masking point of view. This information can influence the final decisions on what targets to attack. After selection, the PVG can assist in determining the optimum approach direction

and altitude to the target. Finally it can provide the capability to conduct a mission rehearsal without risking personnel or aircraft.

The following chapter explains the steps taken toward satisfying the thesis objective. Before the grand capability of converting satellite imagery directly to a PVG database could be realized much had to be learned about the manipulation of database elements to get the desired results. The original PVG, designed primarily to view the Fort Hunter Liggett database, also required some modifications to view the new databases.

III. EXPERIMENTS

The first step toward achieving the thesis objective was the generation of individual objects. This process entailed making modifications to the original PVG. Using the knowledge gained from generating individual objects, imagery and ground contours were then combined to produce three-dimensional simulations. Finally, editing algorithms were developed to enable detailed, real time scene modifications.

A. INDIVIDUAL TREES

The first attempt at database construction was the generation of a flat terrain with an individual tree. For ease of programming the tree was constructed of simple shapes. The tree covered a square of three by three terrain elements. The center element was the trunk. It was assigned an undercover index of zero, an elevation of nine and a dark greyshade. The remaining elements were assigned an undercover index of six, an elevation of nine and a slightly darker greyshade. The resulting object was a cube resting on top of a narrow rectangle, similar to that shown in Figure 3.1.

During the construction of the first tree each element was addressed individually by its x and y coordinates. Next algorithms were developed to accept user input of tree dimensions and a starting point. This input was then used in programming loops to assign all of the elements in the tree automatically.

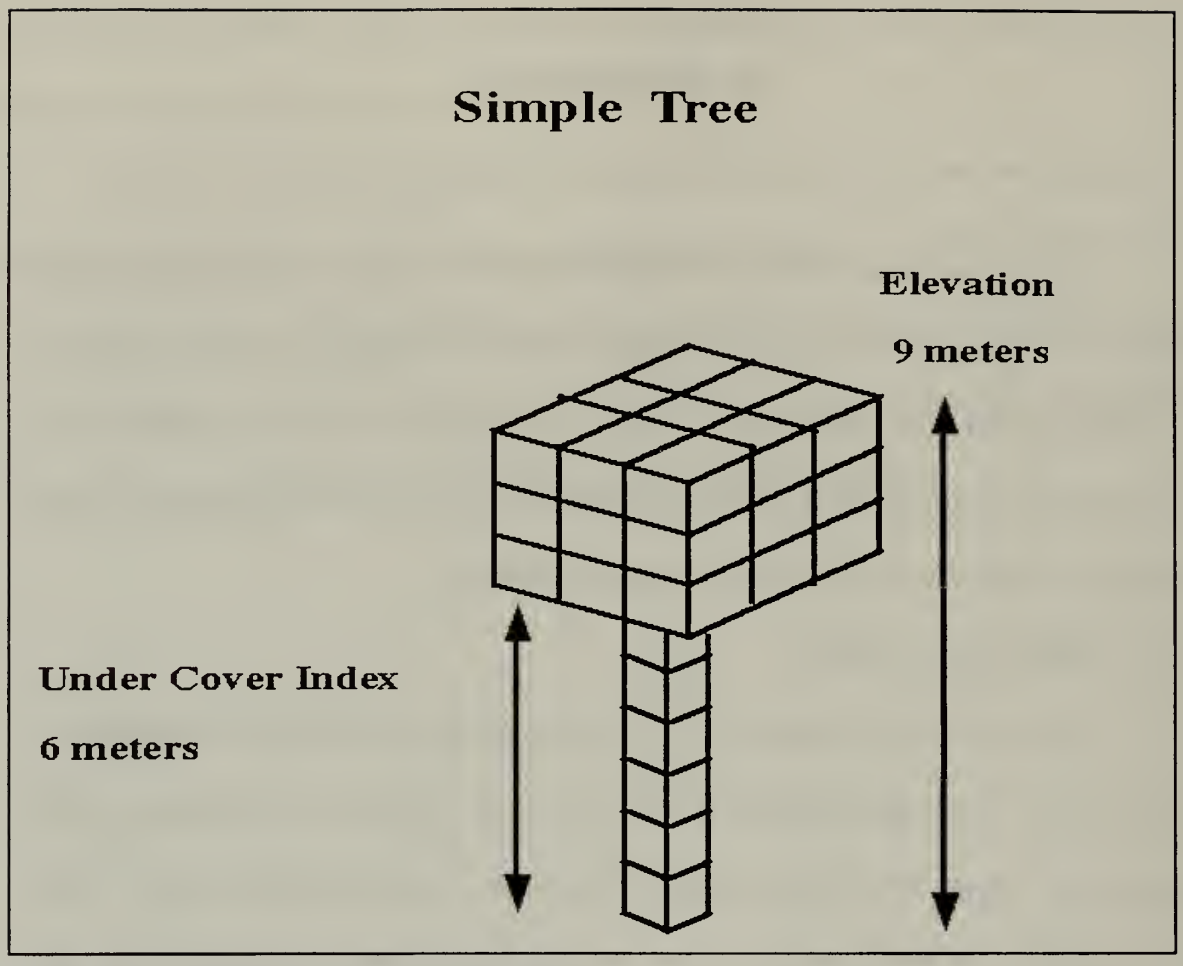

Figure 3.1. Construction of a Simple Tree.

An example tree algorithm is shown in Figure 3.2 Different combinations and sizes of rectangle were tried to make different types of the trees. Figure 3.3 shows examples of the first trees as viewed by the PVG. These trees provided great insight into the manipulation of bit fields, but their appearance is very cartoon-like.

```
for(x = 231; x <= 233; x ++)
 {
  for(y = 231; y <= 233; y ++)
  {
    if(x == 232  &&  y == 232)
    DAT[x][y] =  trunk;
    else

     DAT[x][y] =  canopy;
  }
 }
```

Figure 3.2. Sample Tree Algorithm

Figure 3.3. Simple Trees as Viewed by PVG.

B. INDIVIDUAL BUILDINGS

The next objects to be generated were buildings. To aid in depth perception there was a requirement to assign different greyshades to adjacent walls, edges and the roof. This posed a difficulty because the bit fields in the database only allow for a single greyshade to be assigned to each data element. This database limitation was apparent in the first trees when attempts were made to make the trunk and canopy different colors. The result was a square in the center of the top of the canopy which was the color of the trunk and not that of the rest of the canopy. This limitation is visible in Figure 3.3 and was overcome through the use of the surface normal bit field.

Again for simplicity the first buildings were rectangular with walls parallel to the x and y axes. To distinguish edges, these elements were assigned black. The remaining elements were all assigned the same color grey. In addition the outer elements, which were not at the corners, were assigned a surface normal value. Elements on opposite walls were assigned the same surface normal values and each pair of walls were given slightly different values. Figure 3.4 shows an example of a simple building. Before this building could be displayed however, modifications to the original PVG were required.

C. MODIFICATIONS TO THE PVG

In the original PVG, the main program *pvg-main.c* calls 10 named functions during the course of a run. To meet the objective of this thesis the Main Program, *pvg-main.c* and two functions, *call_map.c* and *init_window.c* required minor changes. Most of the work was

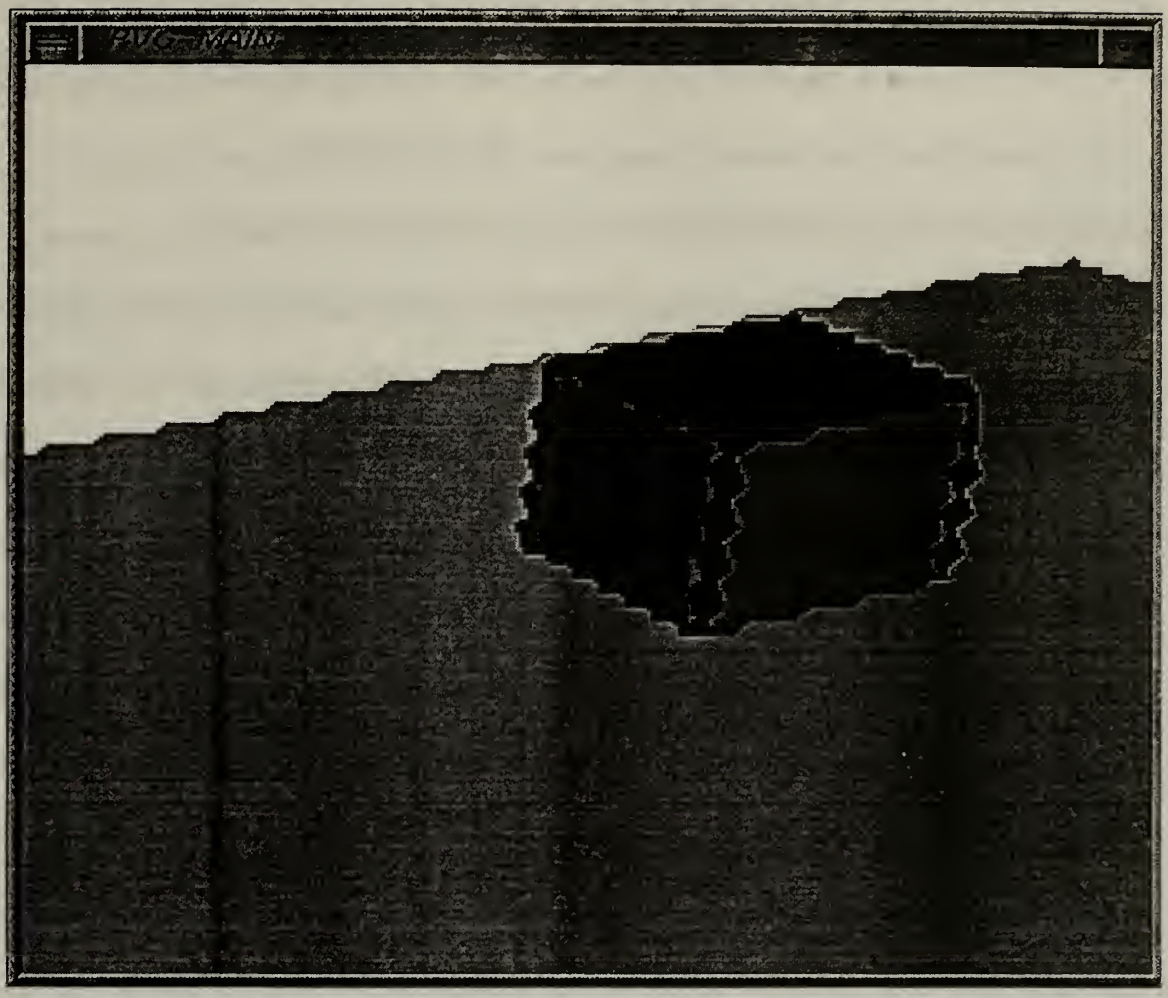

Figure 3.4. Simple Building as Viewed by the PVG.

focused on modifications to the functions *datagen.c*, *zigzag.c* and *hitgrnd.c* and on writing new functions *edit.c* and *trees.c*. The latter modifications and the new functions will be discussed in more detail in the following pages.

1. Ray Trace Algorithm

The efficiency of the ray trace algorithm, found in *zigzag.c*, is critical to the overall performance of the PVG. As mentioned above between 70 and 80 percent of the total run time is spent in ray tracing. Any modifications to this portion of the program must be carefully considered because new commands may be executed thousands of times.

a. Adjusting Ray Trace Back-Step

One of the time saving features of the ray trace that was described in the introduction is the zig-zag wherein tracing does not begin from observer for every ray. Also discussed was the need for a back-step. Moving directly up to the next ray in a column, one may find that the ray under an overhang, but if the ray trace had begun from the observer it would have actually terminated several steps earlier on the overhang itself. This effect was illustrated in Figure 1.7. The dilemma is that starting all ray traces from the observer is computationally expensive and realistic movement is lost, but starting successive traces where the previous one ended sacrifices detail. As a compromise the original PVG uses a back-step equal to five steps back from where the previous ray trace terminated. Five steps proved to be an excellent compromise for the Fort Hunter Liggett database, but when working with the individual trees shown earlier, the overhangs were consistently lost. In order to accommodate these overhangs, the back-step was adjusted and by trial and error it was found that a back-step of 15 steps provided sufficient detail and was still fast enough to maintain realistic movement.

b. Retaining Previous Height

A related problem also arose when viewing buildings. For simplicity the buildings were not given any overhangs. The problem with viewing buildings came about when a decision had to be made as to whether a ray had struck a wall or the roof. This is an important decision to maintain the sharpness of the edges for improved three-dimensional perception. To facilitate the wall or roof decision, a command was placed in the ray trace to retain the trace height at both the previous and current step. Both of these values are then

passed to *hitgrnd.c* where the decision of what to display is made. The logic required for this decision will be explained later in more detail.

c. Boundary Check

The final modification to the ray trace algorithm in *zigzag.c* was to refine the boundary checking routine. In the PVG it is possible that the observer's position is not above the defined terrain. In this case there exists no terrain height to compare to the z coordinate of the ray trace. It was explained in the introduction that when the z coordinate of the ray trace is equal to the height of the terrain the ray trace is terminated. With the observer position over an undefined region there is not enough information for this logic. It is desirable for the ray trace to be able to continue in this case because the orientation of the boresight may be such that the terrain will eventually be seen. It is possible to force the ray trace to continue stepping when the observer's position is undefined and when the ray finally passes over the defined terrain, there is a height available for comparison. Forcing the ray trace to continue in undefined regions however, gives rise to another problem.

What happens if the ray passes over the terrain and does not intersect? If the trace is forced to continue in the undefined regions it will continue in an infinite loop locking the program in that trace. To avoid this problem the entire database is searched in *datagen.c* to find the minimum elevation value. This value is then passed to the *zigzag.c*. Now if the trace is over an undefined region, as long as the z position of the ray is greater than the minimum elevation of the database the trace will continue until either the terrain is intersected or the minimum value is reached. If the minimum value is reached without an intersection the color white is returned at the minimum elevation. Figure 3.5 shows different

boundary cases. Ray number 3 in the figure reaches the minimum elevation without intersecting the terrain.

Along with ensuring the ray is above the minimum terrain elevation the boundary checking routine also makes sure the x and y coordinates of the trace remain within the defined region As long as either the x or the y coordinate of the ray trace lies in the defined region the ray trace will continue until the elevation criteria is met as described above. If however the ray passes completely over the terrain and is well above the minimum elevation there is no point in wasting time continuing that trace down to the minimum elevation. In this case the trace is terminated at the point where the ray passed over the terrain and the color white is returned at the minimum elevation. Ray number 1 in Figure 3.5 passes over the terrain and ray number 2 strikes the terrain.

These boundary conditions allow the observer to move completely around the outside of the terrain while looking in at the defined region. They also give rise to the apparent edges of the terrain. Figure 3.6 shows the boundary check routine described in this section. Appendix A contains a full listing of *zigzag.c*.

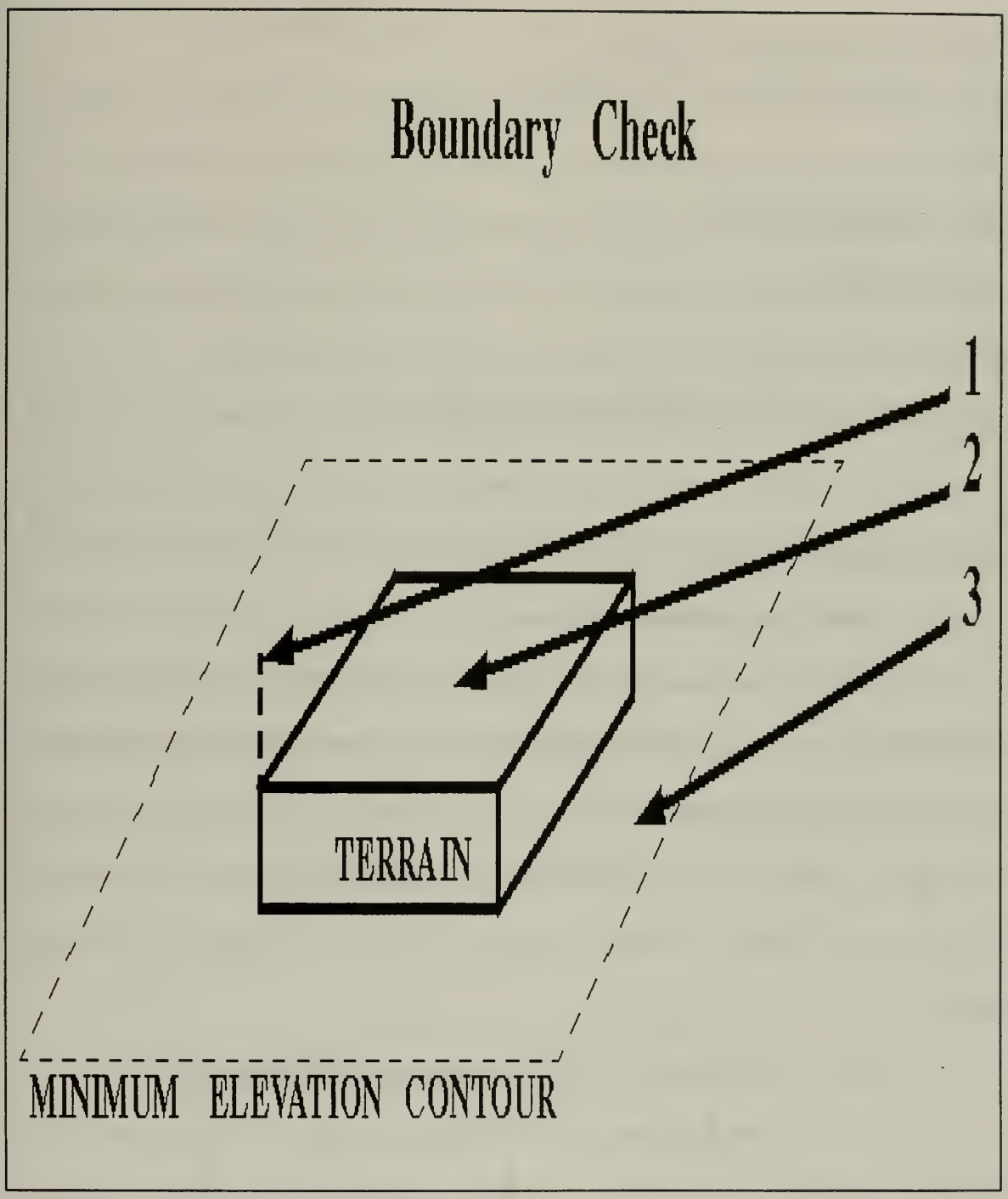

Figure 3.5. Boundary Check Example Cases.

```
If((x <= (float)SCALE) && (x <= 0.0) && (y <= (float)SCALE) && (y <= 0.0))
{
 /* RAY IS OVER THE TERRAIN */
 zterr = (float)((DAT[x][y]&ELEV_MASK)>>21);
}
else
{
 if (zr > (float)ELEV_MIN)
 {
  goto next_step;
 }
 else
 {
  data_flag = 1; /* RAY HAS PASSED OVER TERRAIN */
  break;
 }
}
```

Figure 3.6. Ray Trace Boundary Check.

2. Data Interpretation Algorithm

Once a ray trace intersects the terrain *hitgrnd.c* is called to read and interpret what should be displayed on the screen. The original PVG routines for interpreting natural objects were well established as described earlier. Since there are very few buildings in the Fort Hunter Liggett database they were not addressed separately in the algorithm. Additional logic was required to render buildings. Appendix B contains a listing of the modified *hitgrnd.c.*

a. Distinguishing Between Natural and Man-Made Objects

The first determination made by the modified *hitgrnd.c* is if the ray has struck a natural or man-made object. This determination is accomplished by checking the nature bit. If the nature bit is high then the original algorithm for natural objects is completed. If the nature bit is low then the new routines for man-made objects are invoked.

b. Use of Previous Height to Distinguish Roof from Walls

As mentioned above, to enhance the three-dimensional perception of buildings it is desirable to construct them with walls that are a slightly different color than its roof. The roof should be the greyshade that would be observed from the black and white camera at the observation point. Since the true greyshades of the walls are unknown, arbitrary shades can be chosen as long as they are different from each other and that of the roof.

Along the upper edge of a building there are locations where, depending on the position of the observer and the orientation of the boresight vector, one ray may strike the roof and another may strike a wall at the same (x,y) location. It is therefore necessary to determine whether the roof or the wall was struck. To accomplish this task *hitgrnd.c* uses the previous height information provided from *zigzag.c*. When a ray strikes a building the intersection height is compared to the previous height of the ray. If the previous height was above the elevation at the intersection point, as in the case of ray number 1 in Figure 3.7,

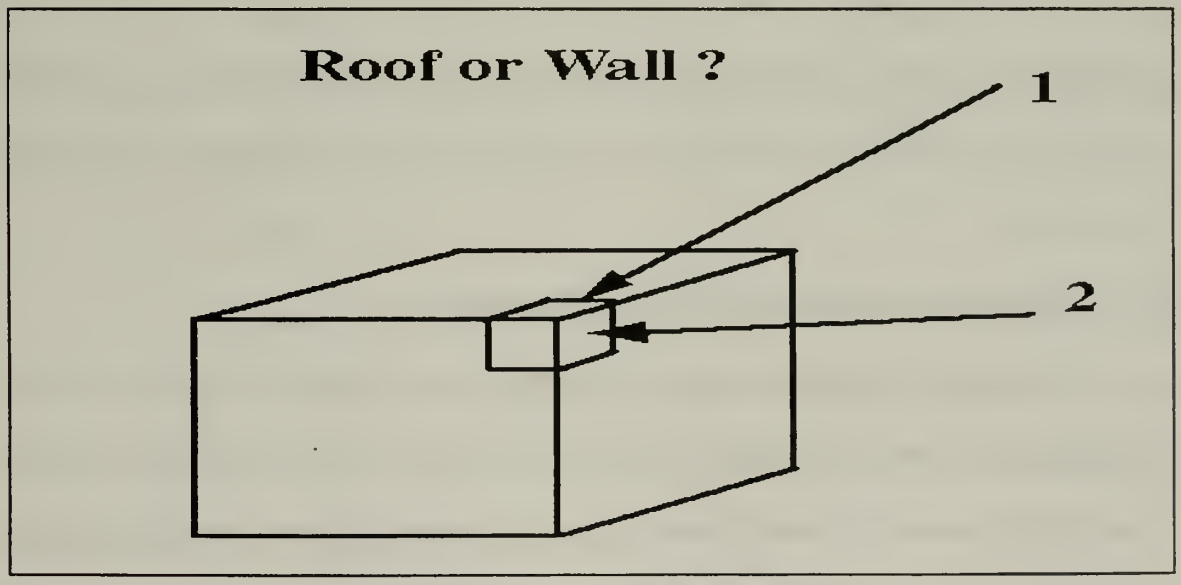

Figure 3.7. Illustration of Roof or Wall Decision Made in *hitgrnd.c*.

then the roof will be seen. If the previous height was below the elevation at the point of intersection, like ray number 2, then a wall will be seen. This logic defines the edge between the roof and the walls.

c. Use of Surface Normal to Distinguish Walls

The distinction between different walls is the next thing to be determined. Since details on the walls, such as windows, doors and greyshades are not available from overhead photographs, it is best to display the walls as single arbitrary greyshades. Speculation on these details has the potential of causing considerable confusion for the user and the additional logic and data requirements would be considerable. Varying the greyshade of adjacent walls however, greatly enhances the three-dimensional perception.

The surface normal bit field is used to make the distinction between adjacent walls. Within the surface normal field there is a north, south, east and west bit. Each wall is assigned one or a combination of these values depending on its orientation. When a wall is hit the total surface normal value is read and a switch statement is used to determine the greyshade of the view. Through proper assignment of surface normals, corners and adjacent walls are distinguished. The method of assigning surface normal values will be described later in detail.

D. TERRAIN INFORMATION CONVERSION

The terrain information used in this project comes from DTED and SPOT data supplied by the Naval Air Warfare Center at China Lake, CA. The information contained in the DTED database is ground contours. The SPOT database contains greyshade data from aerial photographs. The researchers at China Lake have developed a program called

SELECT which combines the DTED and SPOT data to display a two-dimensional image of the terrain with elevation contour over-lays. Using the SELECT program, a portion of the database can be viewed. A modification to the SELECT program was written that writes the elevation data to a file along with its corresponding x and y coordinates. A similar file was also created for the greyshades from the SELECT image.

The DTED elevation information is accurate to every 100 meters. The SPOT greyshade information is accurate to every 30 meters. In order to make these different scales conform to the PVG's one meter scale, data scaling is performed in *datagen.c*. Appendix C contains a listing of *datagen.c*. The DTED elevation values are divided by 100 and the SPOT greyshade values by 30 before they are stored in the data array. Additional greyshade scaling is required for the PVG because the Silicon Graphics is a color machine and the image is black and white. The image intensity is scaled to give a reasonable distribution between black and white. This scaling is performed immediately after the data is read from the files and the scaled values are stored in separate two-dimensional arrays. When these arrays are complete their bit field values are added and nature bit is set high. The final result is placed in DAT[x][y].

E. USER INTERFACE

The conversion of DTED and SPOT data provided an excellent simulation of barren land like mountainous regions with no large vegetation or buildings. Figures 3.8 and 3.9 show the static and dynamic views that are seen from DTED and SPOT data for a section of mountainous terrain in the western United States.

In wooded or urban areas however, the three-dimensional image lacked important elevation details. Since the DTED database only contains ground contours, objects in the scenes do not have any height. Database editing was required to give heights to trees and buildings in the image.

The manipulation of bit fields to generate individual objects described in sections A and B of this chapter was effective but it was also quite tedious when many trees and buildings had to be constructed. With aerial photographs and ground contours combined to form a three-dimensional terrain view, the next step toward the thesis objective was to be able to add large numbers of objects to the terrain in an efficient manner. Ideally the user would then be able to rapidly see the effects of the changes he/she had made.

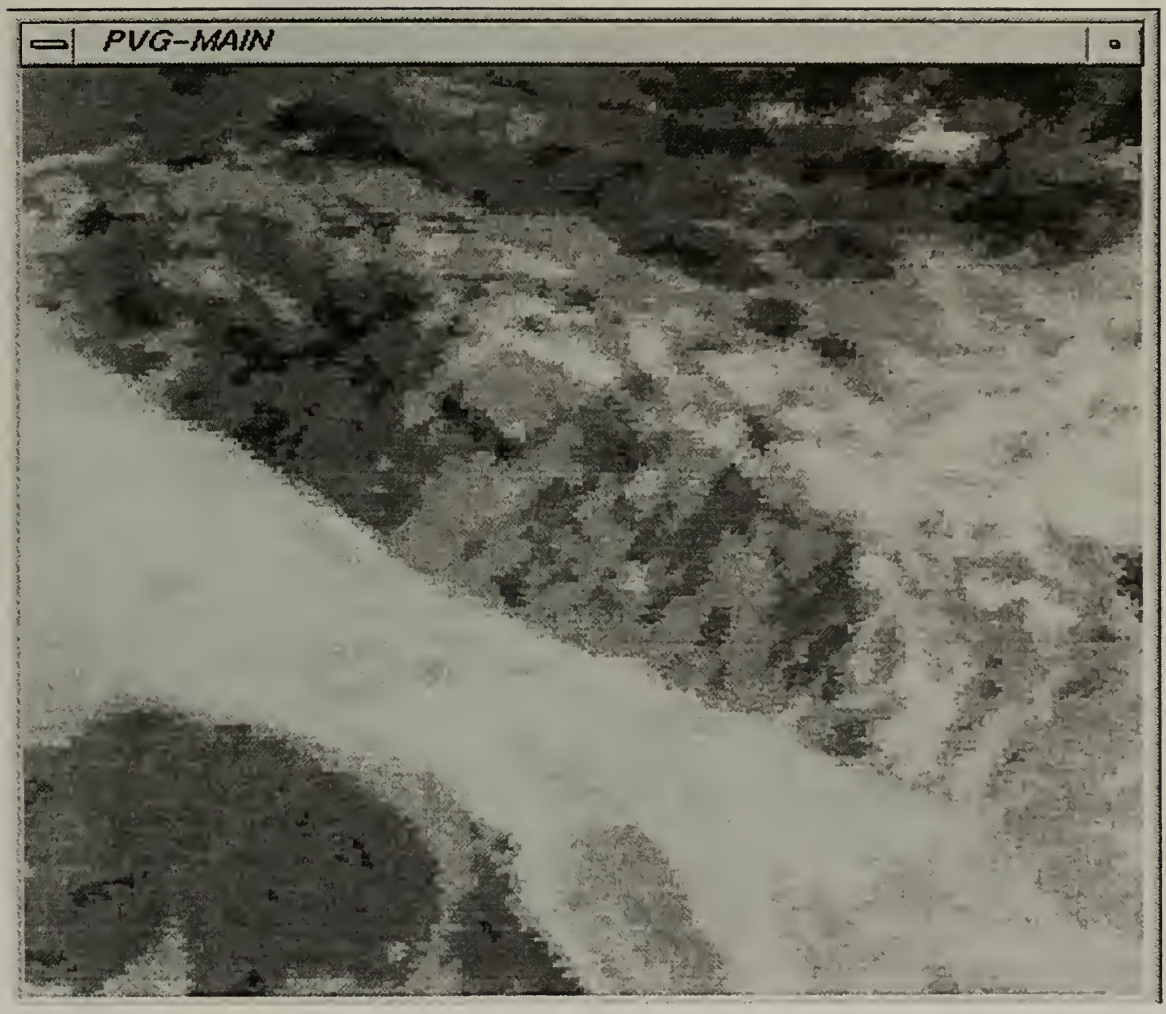

Figure 3.8. Static PVG View of Mountainous Region from DTED and SPOT Data.

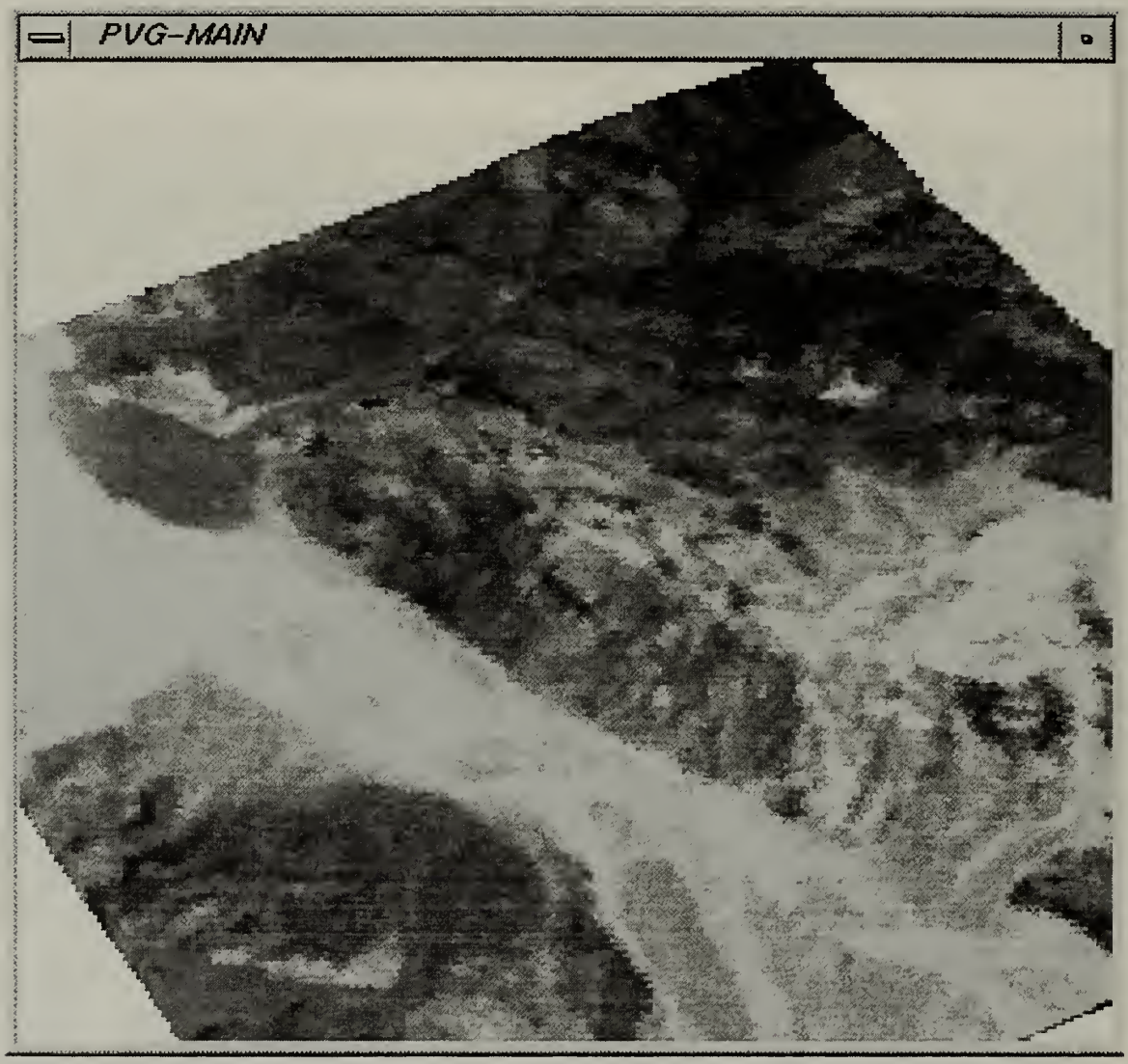

Figure 3.9. Dynamic, Three-Dimensional PVG View of Mountainous Region from DTED and Spot Data.

1. Real Time Editing

Real time editing is the ability to see the effects of an edit procedure rapidly in the dynamic view without having to restart the program. Real time editing is achieved through the use of Silicon Graphics device and queuing commands. A device or button is designated and the computer polls this device looking for a change in its status. A record of status changes is kept in the device queue. When a status change is detected the computer performs a specified task. The device commands are used extensively in the edit process and are particularly important in achieving real time editing. [Ref.3: p.5.1-9.]

The function *edit.c* which is described in the following sections enables the user to make changes to the database that the PVG is viewing. In the interest of conserving computation time it is a separate function that is only called when needed. A listing of *edit.c* can be found in Appendix D.

The device/button used to signal the computer to call *edit.c* is the keyboard control key. With the cursor in the dynamic window whenever the control key is depressed, the edit window is activated and changes can then be made to the database. The control key is polled every time a ray trace intersects the terrain at the beginning of *hitgrnd.c*. The device polling slows the processing of the dynamic view but the frame rate is still acceptable.

2. Making Database Changes

When the modified version of the PVG is run three windows are displayed. Figure 3.10 shows the new PVG screen view. The left window is the edit window and like the original PVG it shows a view of the terrain from an observer looking straight down. The window on the right is the dynamic PVG view and the lower left window is a UNIX Shell

Figure 3.10. Sample Modified PVG Screen Arrangement.

for user interface. The images in Figure 3.10 were generated by the modified PVG using data taken from the DTED and SPOT databases. It shows a small airport near China Lake in California.

Looking at an aerial photograph like the ones shown above, groups of trees and buildings can be identified by there greyshades in the image. In order to make the PVG simulation more realistic more details about objects such as trees and buildings must be incorporated into the database.

With knowledge of the types of trees found in the region, an educated guess can be made as to the average tree height and under cover indices one would expect to see. This

information can then be edited into the database. To avoid the tedious work of addressing each pixel, functions were written to automatically assign values of the tree heights and undercover indices such that they vary in some fashion around the mean value. These varying values are then assigned until the entire area of trees is populated to a specified density.

If addition intelligence is available about actual building heights, this information can be incorporated too. Again a routine was written to automatically construct buildings according to values input by the user. The function *edit.c* enables these database modifications.

a. Marking a Search Area

The first step in the automatic generation of tree stands and buildings is marking the area in the image where these objects are to be created. As mentioned earlier one is able to distinguish trees and buildings in an aerial photograph by their greyshade alone. By isolating areas of different greyshades, image changes can be made to the characteristics of these areas.

In the edit window an area can be identified using the mouse. The areas identified are in the shape of rectangles. By pressing the middle mouse button the user can assign the (x,y) position of the cursor to one vertex of the search rectangle. Next the user selects the diagonal vertex using the middle mouse button again. These two points are then used to section off a rectangular area. Since all objects in the scene are not perfect rectangles, the user must select a rectangle that is large enough to include the entire area of interest.

b. Searching for Similar Adjacent Pixels

Now that a reduced area of the database has been selected it is necessary to search this area to find all adjacent pixels of the same greyshade plus or minus a user defined value. Moving the cursor to a representative pixel and pressing the right mouse button will begin a search, within the defined rectangle, for all pixels adjacent to the one selected and with similar greyshades. As these pixels are found their write bits are set high and their greyshade in the edit window is changed to black so the user can see which pixels were selected. If necessary he/she can refine the search parameters to capture all of the desired area. After an area is selected, *edit.c* allows the user to designate it as a tree stand or as a building. Figure 3.11 shows the edit window with two sections of the terrain selected for editing, shown as the dark areas in the image.

3. Generation of Tree Stands

If the selected area is to be designated as trees, the user presses the T key and a search is started at the lower left of the edit rectangle. The search proceeds to the right across the first row of pixels. The search is looking for the pixels whose write bits are high. When a high write bit is found the function *trees.c* assigns values for height, undercover index and elevation. These characteristics give the appearance of trees in the dynamic view. Appendix E contains a listing of the *trees.c*.

The values assigned by *trees.c* are generated based on user input. The user inputs the average tree height and under cover index for the highlighted region. He/she must also input an acceptable bound around the average values. *Trees.c* then uses the C random number

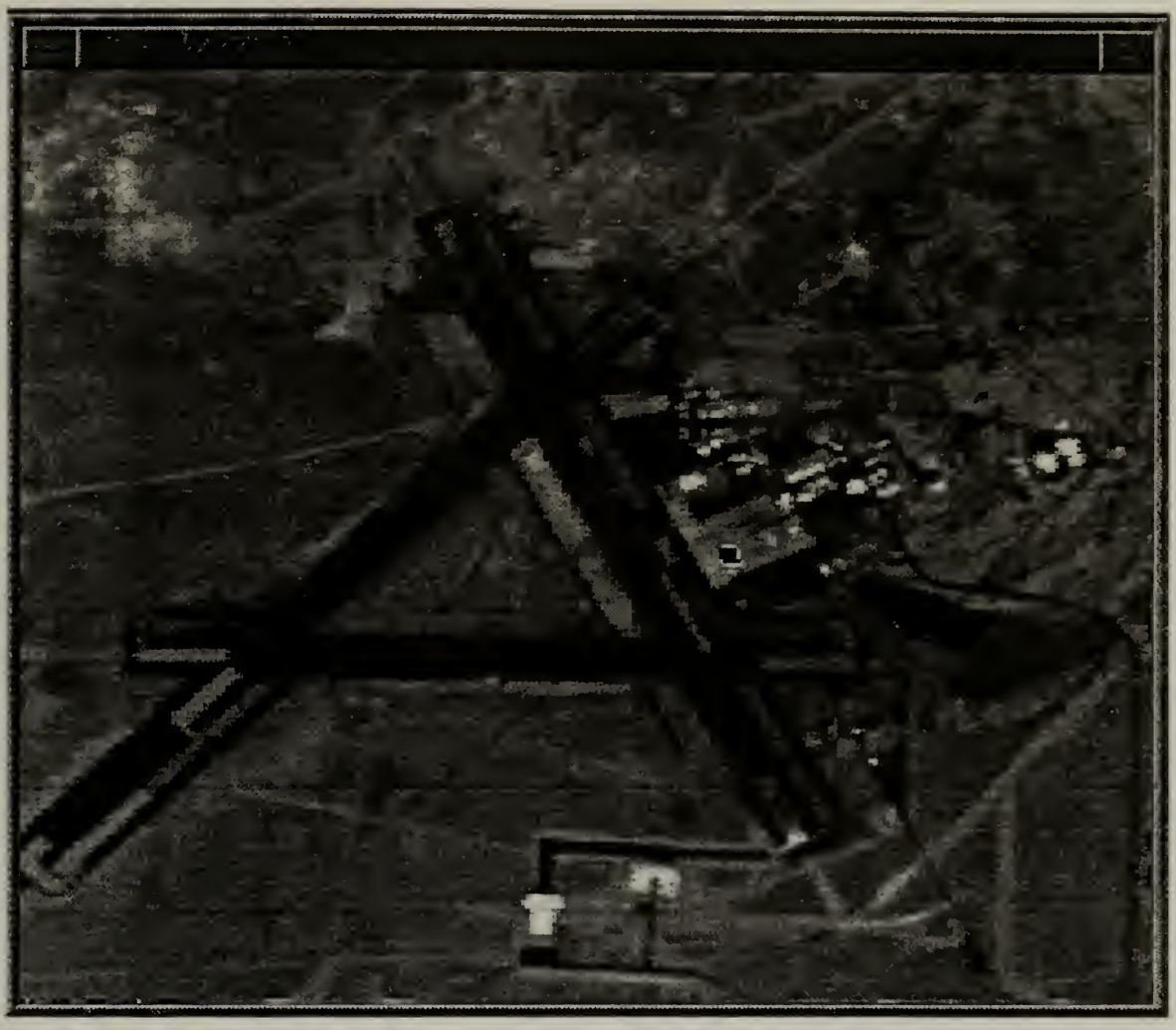

Figure 3.11. Edit Window Showing Two Areas Highlighted for Editing.

generator to generate bounded values for the these characteristics. These values are then encoded for the given (x,y) position. Once the characteristics for the pixel have been set, the write bit is put low and the search moves across each column and up to the next row until the entire area of the edit rectangle has been searched.

4. Generation of Buildings

If the selected area is to be designated a building, the B key is pushed to initiate the building routine. Along with the height of the building, the surface normal values must be assigned to the walls. Like the generation of trees, the building algorithm begins with a

search started from the lower left corner of the edit rectangle. Like before the search routine is looking for pixels whose write bit is high.

As the search moves from left to right, the first pixel encountered with a high write bit is assigned a western surface normal which equals eight. The last pixel encountered in the row with a high write bit is assigned an eastern surface normal which equals four. After each pixel in the row is checked, the search continues on to the next higher row until the entire edit rectangle has been searched.

After the edit rectangle has been checked by row, the search routine returns to the lower left corner and the pixels are checked from bottom to top in each successive column. The first pixel encountered with a high write bit in the vertical search is assigned a southern surface normal equal to two. The last pixel encountered with a high write bit in the column is assigned a northern surface normal value of one. Figure 3.12 shows an example of surface normal assignments for an arbitrary building shape.

When the horizontal and vertical searches are complete the sum of all surfaces normals that were assigned to a pixel are added to the appropriate values of building height, input by the user, and elevation, which is determined from the original terrain data. This information is encoded for the position of the pixel. The write and nature bits are also set low to complete the building process. The technique of assigning surface normal values described above enables a building with any orientation to be displayed with distinct edges and different colored walls. It also enables buildings of many shapes to be rendered. The building routines are contained in *edit.c*.

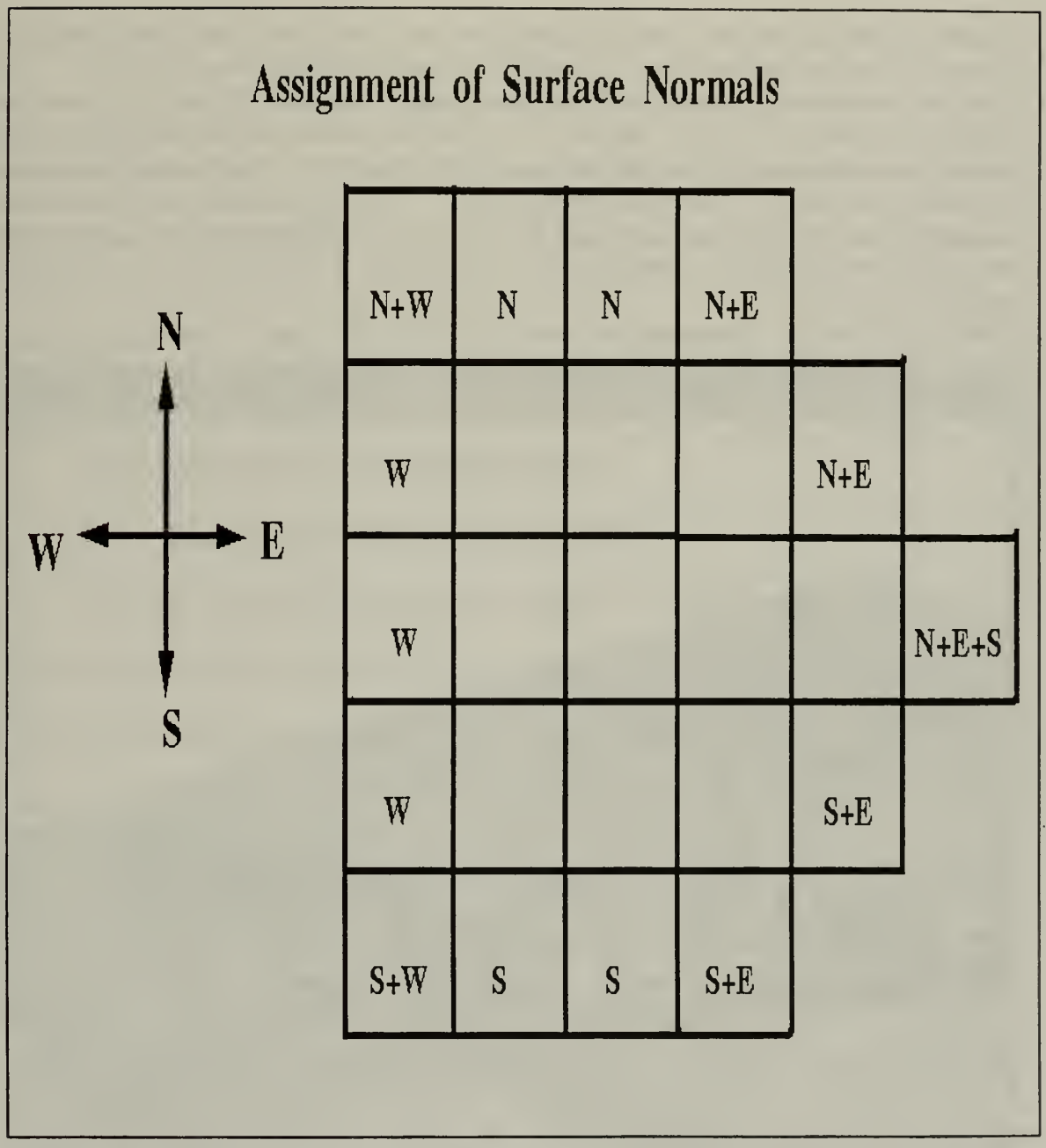

Figure 3.12. Example of Surface Normal Assignments for an Arbitrary Building Shape.

Once all editing is completed the dynamic view can be reactivated by pressing the escape key with the cursor in the edit window. The new changes will appear and can be viewed in three-dimensions in the next image frame. There is no limit to the number of times the user can toggle between the edit and dynamic windows. Figure 3.13 shows the dynamic view that corresponds to the edit window of Figure 3.11. The building option was selected for the edit region on the left and the tree option was selected for the region on the right.

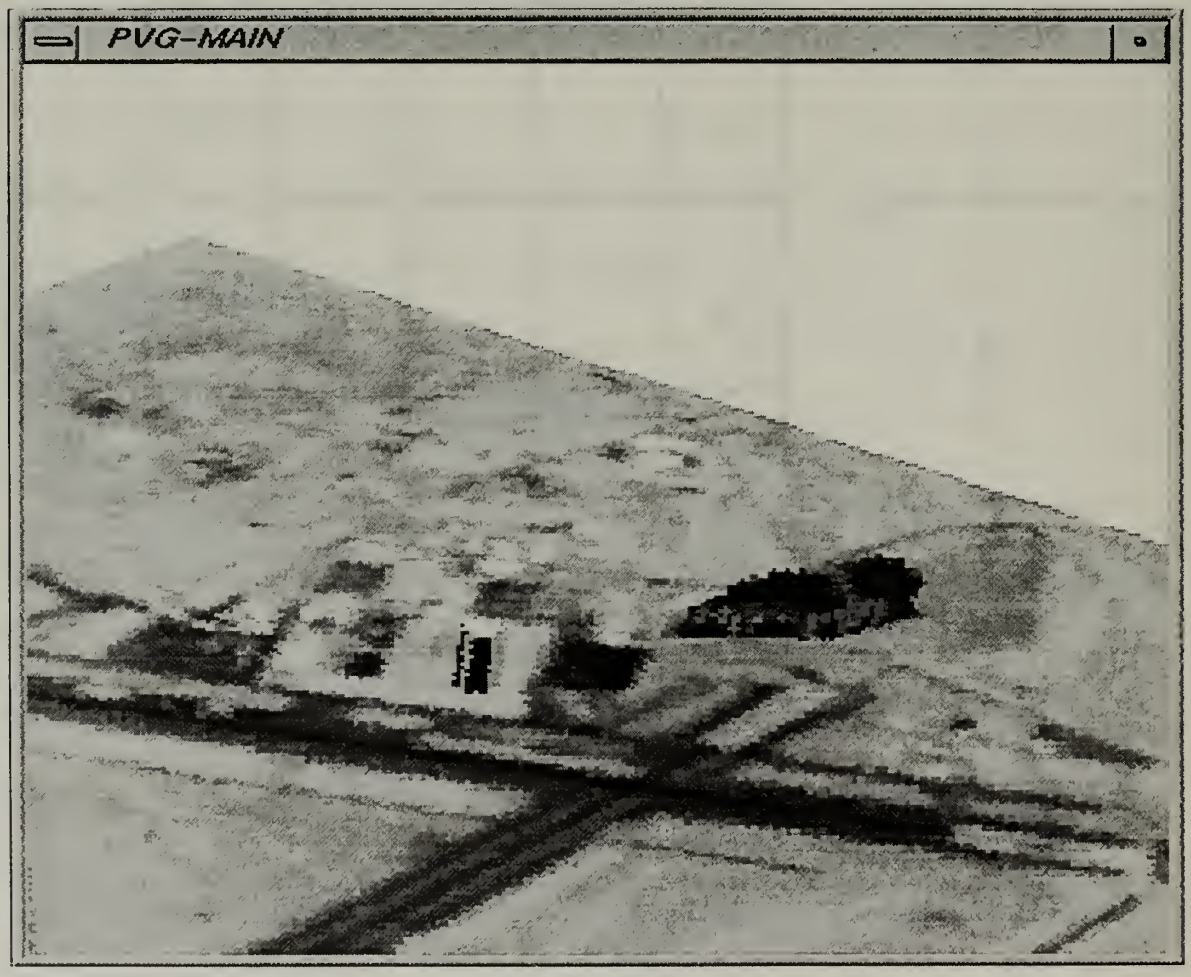

Figure 3.13. Edited PVG View with a Building on the Left and Trees on the Right

IV. DISCUSSION

The objective of this thesis as stated in Chapter 2 was to take an image of an area and some topographical information and generate a database which will result in a realistic simulation when used with the PVG. The computer programs listed in Appendices A through E are able to take SPOT imagery and DTED topographical information and generate a realistic database for use with the PVG--mission accomplished. Although this thesis objective has been met, like any worthwhile project there is more to be done.

A. SUGGESTED FOLLOW-ON RESEARCH

1. Transition from Edit to Full-Speed Mode

In the end, the driving requirement for realism in a synthetic environment scenario is execution speed. The most detailed and complex scene will be unrealistic unless the user is able to move about rapidly. Every building and tree may be exactly sized, precisely placed and correct to the last detail but if the intended users, strike planners and pilots, are unable to perceive approximate mission speeds, the utility of the simulation is greatly diminished. Object masking effects may be observed in a slow moving scenario but mission optimization and realistic mission rehearsals are impossible. The high speed at which events in a strike mission occur dictates that careful planning and considerable training be conducted if the strike is to have precision. Decision making must be rapid and correct. The more speed a synthetic environment scenario can introduce to the planning and especially the rehearsal phases of a mission, the more useful it will be to the military.

In the modified PVG, *edit.c* is necessary for the initial creation of the database. The real time editing feature is very useful for building a realistic scene. The problem with the real time edit feature is that it requires device queuing. The additional logic at the beginning of *hitgrnd.c* means that every time a ray trace strikes an object or the terrain, the program must stop and check to see if the control key has been pressed. This additional logic has a noticeable effect on the execution time of the dynamic view. It is recommended that in the future, the PVG have the option to run in two modes: edit mode and full-speed viewing.

After the scene has been edited and is satisfactory for the given mission, an option in *edit.c* could allow the user to save the completed database to a file. With the database in a file, a version of the PVG without the edit functions can be used to view the scene. Possibly another device queue could be added to *edit.c* which can activate the full-speed mode.

2. Refine Edit Process

It is the opinion of the author that there are an infinite number of possibilities for user input during the edit process. Refining the edit process can make it more capable and more user friendly. Included in this section are some ideas for future improvements.

a. Marking Edit Areas

Currently *edit.c* allows users to section off a rectangular area in the edit window for adding details. The ability to overlay the rectangle on the image would greatly improve this feature. A visible outline of the edit area would make it easier for the user to ensure the entire area of interest is captured in the first try. Examples of line drawing algorithms for the Silicon Graphics are available in Reference 3.

Another possibility is to allow any arbitrary area, instead of just rectangles, to be specified so that the actual object shapes can be traced. The ability to accurately outline the edit area can eliminate the need to search for similarly shaded adjacent pixels as in *edit.c*. This pixel search was used to compensate for the fact that arbitrarily shaped objects could not be outlined. If the edit area can be precisely controlled, it can be drawn so that it includes only similarly shaded pixels.

Another possibility is creating a paintbrush function where the mouse is used to paint in trees or buildings. This feature would allow for touch-ups to the bulk generation methods already used. Perhaps a small number of pixels need to be adjusted after a large tree stand has been generated, a paintbrush function would be useful for that task.

b. Undo/Redo

A common edit feature is the *undo* or *redo* option where the user can add or remove the last edit command in a minimal number of steps. If an area has been inadvertently identified as a building when it should have been trees, the *undo/redo* option could be used to rapidly correct the discrepancy. This time saving feature would be extremely useful because most discrepancies will probably not be apparent until the user escapes out of the edit window and begins to move in the dynamic window.

c. Improved Tree Distributions

In *trees.c* the placement and sizes of trees are based on the C random number generator algorithm. More sophisticated algorithms can be written to approximate other statistical distributions. These could then be used to distribute tree characteristics in an attempt to more closely represent nature. For example the heights of trees could be

distributed as a Gaussian and their placement could be as a Triangular distribution. Different distributions could be tested to see which most accurately represents the area in the image.

d. Refined Surface Normal Assignment and Interpretation

The building routine in *edit.c* results in the possibility of 12 different surface normal values being assigned in all or any given building. This large number of possible values drives the need for the extensive SWITCH statement in *hitgrnd.c*. Any wall that is not parallel to the x or y axes will have a jagged appearance because of the one meter scale of the PVG. Since each data element is considered in the horizontal and vertical searches currently used to assign surface normals, an off axis wall may have many different surface normal combinations assigned to each element. If off axis walls could be identified, all the elements in the wall could then be assigned the same surface normal and the horizontal and vertical searches could be avoided.

B. TARGET MOTION

Target motion algorithms have been developed for the original PVG. [Ref. 1: p.14-15] Incorporating similar concepts so that they can be used in the new databases could enable different planning and training scenarios. Actual target motion could be anticipated and programmed into the simulation.

V. CONCLUSION

Simulations for planning and training have already proven their usefulness in the United States Armed Forces.The PVG is an example of a simulation with excellent opportunities for military application. While working on this thesis many insights were gleaned about how the PVG functions and the creation of realistic databases. This insight was applied directly to make enhancements to the PVG and also openned up new avenues for continued research. PVG enhancements made as a result of this thesis include:

- Algorithms for the generation of individual and groups of objects on a terrain.
- Modifications to PVG functions to enable viewing of generated objects.
- Conversion of DTED and SPOT data into a format usable by the PVG.
- Algorithms to allow real time editing of a database being viewed by the PVG.

As the face of the military and modern warfare changes, simulations of ever increasing complexity will be required to adequately prepare for future adversaries. Battlefield simulations can provide valuable insight into mission planning and rehearsal. This insight will directly translate into saved lives, saved equipment and saved money. In today's high tech warfare, having the sun at your back is no longer enough of an edge, synthetic environments can provide a high tech edge for the warriors of today and the future.

APPENDIX A. ZIGZAG.C

```
/*
  PROGRAM:  zigzag.c
  Basic ray trace algorithm.

  modified:  25SEP96
*/

#include<stdio.h>
#include<stdlib.h>
#include <math.h>
#include <gl/gl.h>
#include <gl/device.h>
#include"PVG_DEC.IN"

extern float ddcos[3][3];
extern long DAT[SCALE][SCALE];
extern int IFOVNOW[10], ELEV_MIN;

int RAY_FLAG, VIEW[PVG_WIDTH*PVG_HEIGHT];

void zigzag(int, int);

void zigzag(int cstart, int cend)
{
void edit();

 int hitgrnd(int ei, int ni, float zr, float zprev, float zterr, int step);

 register int    step, stepmin;
 register float  e, n, zr, zprev, de, dn, dz, zterr;

 int  col, ray_num, i, j, data_flag=0, red_flag=0;

 float idcos20, idcos21, idcos22;

 const float deb = ddcos[0][0];              /* boresight direction cosines */
 const float dnb = ddcos[0][1];
 const float dzb = ddcos[0][2];
 const float dcos10 = ddcos[1][0];
```

```
const float dcos11 = ddcos[1][1];
const float dcos12 = ddcos[1][2];
const float dcos20 = ddcos[2][0];
const float dcos21 = ddcos[2][1];
const float dcos22 = ddcos[2][2];

RAY_FLAG = step = stepmin = ray_num = 0;  /* initialize these variables */

e  = (float)IFOVNOW[0];                   /* define start location */
n  = (float)IFOVNOW[1];
zr = (float)IFOVNOW[2];

for(i=cstart; i<cend; i++)
 {
  idcos20 = (float)i*dcos20;   /* calc. these as few times as nec. */
  idcos21 = (float)i*dcos21;
  idcos22 = (float)i*dcos22;

  j = 0;                                  /* start row loop */

  do
   {
    de = deb + (float)j*dcos10 + idcos20;
    dn = dnb + (float)j*dcos11 + idcos21;
    dz = dzb + (float)j*dcos12 + idcos22;

    do
     {
next_step:
      step++;
      zprev = zr;  /*USED FOR BUILDINGS*/
      zr    = (float)IFOVNOW[2] + (float)step*dz;
      e     = (float)IFOVNOW[0] + (float)step*de;
      n     = (float)IFOVNOW[1] + (float)step*dn;

/******* BEGIN BOUNDARY CHECK *******/

if((e<=(float)SCALE) && (e>=0.0) && (n<=(float)SCALE) && (n>=0.0))
 {
  /* RAY IS OVER THE TERRAIN */
  zterr  = (float)((DAT[(int)e][(int)n]&ELEV_MASK)>>21);
 }
else
```

```
 {
  if(zr>(float)ELEV_MIN)
   {
    goto next_step;
   }
  else
   {
    data_flag = 1;
    break;
   }
  }
/******* END BOUNDARY CHECK *******/

     } while (zr > zterr);

    if ( data_flag )
     {                    /* out of bounds test */
      col = 255;
      goto v_set;
     }

    col = hitgrnd((int)e, (int)n, zr, zprev, zterr, step);

    if(RAY_FLAG)
     {
      RAY_FLAG=0;
      goto next_step;
     }

/* draw 256x256 white grid

    if (((int)e%256==0) || ((int)n%256==0))
     col = 255;
*/
/*
 * unexpected generation of VIEW to suit display routine
 * sets color of target or terrain GSV
*/

v_set:
    VIEW[(j*PVG_WIDTH+i)] = 65793*col*3;
    data_flag = 0;                    /* clear flags */
    step -= 15;
```

```
  } while (++j < PVG_HEIGHT);    /* end of column loop */

  step = 0;
  }                /* end of row loop */

if (ray_num>0) {fprintf(stderr, "ray hits = %d \n", ray_num);}
}
```

APPENDIX B. HITGRND.C

```
/*
  PROGRAM hitgrnd.c
  Data interpretation algorithms.

  modified:  29OCT96
*/

#include<stdio.h>
#include<stdlib.h>
#include <math.h>
#include <gl/gl.h>
#include <gl/device.h>
#include"PVG_DEC.IN"

int hitgrnd(int ei, int ni, float zr, float zprev, float zterr, int step)
{
 extern long DAT[SCALE][SCALE];
 extern int IFOVNOW[10], RAY_FLAG;

 int shadow, col;
 float zter;

qdevice(CTRLKEY);

if(getbutton(CTRLKEY)==TRUE)
 edit();
else
{
 /* Check if natural of man-made */

 if(((DAT[ei][ni]&NATURE_MASK)>>7)==1)
 {
                          /* Natural objects */
  shadow=0;
                          /*Check for bald terrain*/

  if ((DAT[ei][ni]&UCI_MASK)==0)
  {
   col=(DAT[ei][ni]&GSV_MASK);
```

```
 return((int)(col+(0.25*col)));
 }
                    /*Overhang, check if terrain is hit*/

zter=zterr-(float)((DAT[ei][ni]&VGH_MASK)>>10);

if(zr<zter)
{
 col=shadow;
 return(col);
 }
                /*If not, check if we are in the overhang*/

if(zr>(zter+ ((float)((DAT[ei][ni]&UCI_MASK)>>18))))
{
 col=(DAT[ei][ni]&GSV_MASK);
 return((int)(col+(0.25*col)));
 }

 RAY_FLAG=1;
 return(255);
}
else
{
    /* Man-made objects */

 if(zprev >= zterr)
  {
    /* Roof hit */
  col=(DAT[ei][ni]&GSV_MASK);
  return((int)(col+(0.25*col)));
  }
 else
     /* Wall hit */
 {
  switch (((DAT[ei][ni]&SURF_MASK)>>14))
  {
   case 1:
   {
    col = 31;
    return(col);
   }
   case 2:
```

```
    {
     col = 31;
     return(col);
    }
    case 4:
    {
     col = 41;
     return(col);
    }
    case 5:
    {
     col = 21;
     return(col);
    }
    case 6:
    {
     col = 21;
     return(col);
    }
    case 8:
    {
     col = 41;
     return(col);
    }
    case 9:
    {
     col = 21;
     return(col);
    }
    case 10:
    {
     col = 21;
     return(col);
    }
   }
  }
 }
}
```

APPENDIX C. DATAGEN.C

```
/*
  PROGRAM:  datagen.c
  DTED and SPOT data conversion algorithms.

  25SEP96
*/

#include<stdio.h>
#include<stdlib.h>
#include"PVG_DEC.IN"

int RES;
long int DAT[SCALE][SCALE], ELEV_MIN;
unsigned long RVIEW[SCALE*SCALE];

datagen()
{
 FILE *fp;

 int i, j, k, kprev, grey[SCALE][SCALE], kscale, shift, natural,
    ucin, height, write;
 float kfscale;

 kprev    = 100000;
 natural  = 1<<7;
 ucin     = 0<<18;
 height   = 0<<10;
 write    = 0<<6;

/*Get greyshade data*/

 fp=fopen("aport_spot.dat","r");

 for(i=255;i>=0;i--)
 {
  for(j=0;j<256;j++)
  {
   fscanf(fp,"%d",&k);
   grey[j][i] = (int)(0.25*k);
```

```
  RVIEW[i*SCALE+j]=k*65793;
 }
}

fclose(fp);

/*Get elevation data*/

fp=fopen("aport_elev.dat","r");

for(i=255; i>=0; i--)
{
 for(j=0;j<256;j++)
 {
  fscanf(fp,"%d",&k);
  kfscale  = (2.0*k)/30.0;
  kscale   = (int)kfscale;

  if(kscale<kprev)
   ELEV_MIN = kscale;

  shift    = kscale<<20;
  DAT[j][i] = shift+grey[j][i]+natural+ucin+height+write;
  kprev    = kscale;

/*
printf("%d  %d  %d  %d  %f %d\n",j,i,grey[j][i],k,kfscale,kscale);
*/
 }
}
fclose(fp);

RES = 1;
}
```

APPENDIX D. EDIT.C

```
/*
  PROGRAM:  edit.c
  User interface algorithms.

  14OCT96
*/

#include<stdio.h>
#include<gl/gl.h>
#include<stdlib.h>
#include <gl/device.h>
#include"PVG_DEC.IN"

#define X 0
#define Y 1
#define XY  2

extern long DAT[SCALE][SCALE], EDITWIN_ID;
extern unsigned long RVIEW[SCALE*SCALE];

extern int elev, height, ucin, nature, written;

void edit()
{

 void TREES(int elevi, float vaveh, float sigmah, int vaveu, int sigmau);

 short mval[XY], lastval[XY], val, xi, yi, x, y;
 Device mdev[XY];
 long org[XY], size[XY];
 Boolean run;

 float vaveh, sigmah, count;

 int shade, band, cup, cdown, shadei, elevi, ucini, heighti, naturei, writteni,
    script, nh, vaveu, sigmau, nu, xI, xII, xA, xB, xC, xD, yI, yII, yA, yB,
    yC, yD, hit, xleft, xright, man, north, south, east, west, ylow, yhigh,
    surfi, bheight, elevbuild;
```

```
view:
 winset(EDITWIN_ID);
 rectzoom(2.0,2.0);
 lrectwrite(0,0,SCALE-1,SCALE-1,RVIEW);

 cup   = 1;
 cdown = 1;
 count = 1.0;

/* Eventually for user input at prompt*/
 band   = 8;
 vaveh  = 5.0; /*average tree height in meters*/
 sigmah = 1.0;  /*standard deviation in tree height*/
 vaveu  = 3;    /*average distance from ground to bottom of canopy*/
 sigmau = 1;    /*standard deviation of under cover index*/
/****/

 qdevice(LEFTMOUSE);/*Get values at cursor position*/
 qdevice(MIDDLEMOUSE);/*Rectangle vertices*/
 qdevice(RIGHTMOUSE);/*Mark desired area inside of rectangle*/
 qdevice(TKEY);/*Make marked area trees*/
 qdevice(BKEY);/*Make marked area building*/
 qdevice(ESCKEY);/*Quit edit and proceed to PVG*/

 getorigin(&org[X], &org[Y]);
 getsize(&size[X], &size[Y]);
 mdev[X] = MOUSEX;
 mdev[Y] = MOUSEY;
 run = TRUE;
 while(run)
 {
  switch (qread(&val))
  {
   case LEFTMOUSE:
   if(val == 1)
    {
     if (getbutton(LEFTMOUSE) == TRUE)
      {
       getdev(XY, mdev, mval);
       mval[X] -= org[X];
       mval[Y] -= org[Y];
       if (mval[X]<0 || mval[X] >= size[X] || mval[Y]<0 || mval[Y] >= size[Y])
        continue;
```

```
    else
     x       = (int)(mval[X]/2.0);
     y       = (int)(mval[Y]/2.0);
     shadei   =  DAT[x][y]&GSV_MASK;
     surfi    = (DAT[x][y]&SURF_MASK)>>14;
     elevi    = (DAT[x][y]&ELEV_MASK)>>20;
     ucini    = (DAT[x][y]&UCI_MASK)>>18;
     heighti  = (DAT[x][y]&VGH_MASK)>>10;
     naturei  = (DAT[x][y]&NATURE_MASK)>>7;
     writteni  = (DAT[x][y]&WRITE_MASK)>>6;

     printf("\rX= %d  Y= %d s= %d s/n= %d e= %d h= %d u= %d n= %d w= %d\n",
      x,y,shadei,surfi,(elevi/2),heighti,ucini,naturei,writteni);
    }
   }
   break;

case MIDDLEMOUSE:
if(val == 1)
 {
  if (getbutton(MIDDLEMOUSE) == TRUE)
   {
    getdev(XY, mdev, mval);
    mval[X] -= org[X];
    mval[Y] -= org[Y];

    if (mval[X]<0 || mval[X] >= size[X] || mval[Y]<0 || mval[Y] >= size[Y])
     continue;
    else
     {
     x       = (int)(mval[X]/2.0);
     y       = (int)(mval[Y]/2.0);
     }
    if(count<=1.0)
    {
     xI = x;
     yI = y;
     printf("Choose the diagonal vertice of the rectangle\n");
     printf("xI= %d  yI= %d\n",xI,yI);
     count = 2.0;
     break;
    }
    if(count>=1.5)
```

```
  {
   xII = x;
   yII = y;

   if (xI<xII)
   {
    xA = xI;
    xD = xI;
    xB = xII;
    xC = xII;
   }
   else
   {
    xA = xII;
    xD = xII;
    xB = xI;
    xC = xI;
   }

   if (yI<yII)
   {
    yA = yII;
    yD = yI;
    yB = yII;
    yC = yI;
   }
   else
   {
    yA = yI;
    yD = yII;
    yB = yI;
    yC = yII;
   }
   printf("The vertices of the chosen rectangle are:\n");
   printf("xII= %d  yII= %d\n",xII,yII);
   printf("(%d,%d) (%d,%d) (%d,%d) (%d,%d)\n",xA,yA,xB,yB,xC,yC,xD,yD);
   count = 1.0;
  }
 }
}
 break;
```

```
  case RIGHTMOUSE:
  if(val == 1)
   {
    if (getbutton(RIGHTMOUSE) == TRUE)
     {
      getdev(XY, mdev, mval);
      mval[X] -= org[X];
      mval[Y] -= org[Y];
      if (mval[X]<0 || mval[X] >= size[X] || mval[Y]<0 || mval[Y] >= size[Y])
       continue;
      else
       xi      = (int)(mval[X]/2.0);
       yi      = (int)(mval[Y]/2.0);
       shadei   = DAT[xi][yi]&GSV_MASK;
       elevi    = (DAT[x][y]&ELEV_MASK);
       ucini    = (DAT[x][y]&UCI_MASK);
       heighti  = (DAT[x][y]&VGH_MASK);
       naturei  = (DAT[x][y]&NATURE_MASK);

       x       = xi;
       y       = yi;
       shade    = shadei;
       script   = 1<<6;

printf("\rIntial values\n");
printf("X= %d  Y= %d SHADE= %d\n",x,y,shadei);

start:
      do                /*check to right of initial point*/
      {
       /*set parameters*/
       DAT[x][y]       = shade+elevi+ucini+heighti+naturei+script;
       RVIEW[y*SCALE+x] = 0xf00;
       x             = x+1;
       shade          = DAT[x][y]&GSV_MASK;
       elevi          = (DAT[x][y]&ELEV_MASK);
       ucini          = (DAT[x][y]&UCI_MASK);
       heighti        = (DAT[x][y]&VGH_MASK);
       naturei        = (DAT[x][y]&NATURE_MASK);
      }while((shade<(shadei+band)) && (shade>(shadei-band))
           && (x>=xD) && (x<=xB) && (y>=yD) && (y<=yB));

      x      = xi-1;
```

```
shade  = DAT[x][y]&GSV_MASK;
elevi  = (DAT[x][y]&ELEV_MASK);
ucini  = (DAT[x][y]&UCI_MASK);
heighti = (DAT[x][y]&VGH_MASK);
naturei = (DAT[x][y]&NATURE_MASK);

/*check to left of initial point*/
while((shade<(shadei+band)) && (shade>(shadei-band))
     && (x>=xD) && (x<=xB) && (y>=yD) && (y<=yB))
{
 /*set parameters*/
 DAT[x][y]      = shade+elevi+ucini+heighti+naturei+script;
 RVIEW[y*SCALE+x] = 0xf00;
 x            = x-1;
 shade          = DAT[x][y]&GSV_MASK;
 elevi          = (DAT[x][y]&ELEV_MASK);
 ucini          = (DAT[x][y]&UCI_MASK);
 heighti        = (DAT[x][y]&VGH_MASK);
 naturei        = (DAT[x][y]&NATURE_MASK);
}

y      = yi+cup;  /*check rows above initial point*/
cup    = cup+1;
x      = xi;
shade  = DAT[x][y]&GSV_MASK;
elevi  = (DAT[x][y]&ELEV_MASK);
ucini  = (DAT[x][y]&UCI_MASK);
heighti = (DAT[x][y]&VGH_MASK);
naturei = (DAT[x][y]&NATURE_MASK);

if((shade<(shadei+band)) && (shade>(shadei-band))
   && (x>=xD) && (x<=xB) && (y>=yD) && (y<=yB))
{
 /*set parameters*/
 DAT[x][y]      = shade+elevi+ucini+heighti+naturei+script;
 RVIEW[y*SCALE+x] = 0xf00;
 goto start;
}
else
{
 y      = yi-cdown;  /*check rows below initial point*/
 cdown  = cdown+1;
 x      = xi;
```

```
      shade   = DAT[x][y]&GSV_MASK;
      elevi   = (DAT[x][y]&ELEV_MASK);
      ucini   = (DAT[x][y]&UCI_MASK);
      heighti = (DAT[x][y]&VGH_MASK);
      naturei = (DAT[x][y]&NATURE_MASK);

      if((shade<(shadei+band)) && (shade>(shadei-band))
         && (x>=xD) && (x<=xB) && (y>=yD) && (y<=yB))
      {
       /*set parameters*/
       DAT[x][y]       = shade+elevi+ucini+heighti+naturei+script;
       RVIEW[y*SCALE+x] = 0xf00;
       goto start;
      }
      else
      {
       cup   = 1;
       cdown = 1;
       goto view;
      }
     }
    }
    break;

  case TKEY:
  if(val == 1)
  {
   if (getbutton(TKEY) == TRUE)
   {
    x       = xD;
    y       = yD;
    shade   = DAT[x][y]&GSV_MASK;
    elevi   = (DAT[x][y]&ELEV_MASK)>>20;
    writteni = (DAT[x][y]&WRITE_MASK)>>6;

starti:
    do
    {
     if(((DAT[x][y]&WRITE_MASK)>>6)==1)
     {
      TREES(elevi,vaveh,sigmah,vaveu,sigmau);
      /*set parameters*/
```

```
   DAT[x][y]       = shade+elev+ucin+height+nature+written;
   }
   x           = x+1;
   shade        = DAT[x][y]&GSV_MASK;
   elevi        = (DAT[x][y]&ELEV_MASK)>>20;
   writteni     = (DAT[x][y]&WRITE_MASK)>>6;
  }while(x<=xB && y<=yB);

  y        = y+1;  /*check rows above initial point*/
  x        = xD;
  shade    = DAT[x][y]&GSV_MASK;
  elevi    = (DAT[x][y]&ELEV_MASK)>>20;
  writteni  = (DAT[x][y]&WRITE_MASK)>>6;

  if(y>=yB)
   break;
  goto starti;
  }
 }
break;

 case BKEY:
 if (val == 1)
 {
  if (getbutton(BKEY) == TRUE)
  {
   bheight  = 10;
   man      = 0<<7;
   script   = 0<<6;
   north    = 1<<14;
   south    = 2<<14;
   east     = 4<<14;
   west     = 8<<14;
   hit      = 0;
   xleft    = xB;
   ylow     = yB;
   x        = xD;
   y        = yD;

   while(y<=yB)
   {
    for(x=xD;x<=xB;x++)
    {
```

```
printf("x= %d y= %d\n",x,y);
      if(((DAT[x][y]&WRITE_MASK)>>6)==1)
      {
       hit = hit+1;

       if(x<=xleft)
        xleft  = x;
       else
        xright = x;
      }
     }

     if(hit>1)
     {
      shade       =  DAT[xleft][y]&GSV_MASK;
      ucini       = (DAT[xleft][y]&UCI_MASK);
      heighti     = (DAT[xleft][y]&VGH_MASK);
      elevi       = (DAT[xleft][y]&ELEV_MASK);
      writteni    = (DAT[xleft][y]&WRITE_MASK);
      DAT[xleft][y] = shade+elevi+ucini+heighti+man+writteni+west;

      shade       =  DAT[xright][y]&GSV_MASK;
      ucini       = (DAT[xright][y]&UCI_MASK);
      heighti     = (DAT[xright][y]&VGH_MASK);
      elevi       = (DAT[xright][y]&ELEV_MASK);
      writteni    = (DAT[xright][y]&WRITE_MASK);
      DAT[xright][y] = shade+elevi+ucini+heighti+man+writteni+east;
     }

     if(hit=1)
     {
      shade       =  DAT[xleft][y]&GSV_MASK;
      ucini       = (DAT[xleft][y]&UCI_MASK);
      heighti     = (DAT[xleft][y]&VGH_MASK);
      elevi       = (DAT[xleft][y]&ELEV_MASK);
      writteni    = (DAT[xleft][y]&WRITE_MASK);
      DAT[xleft][y] = shade+elevi+ucini+heighti+man+writteni+west;
     }

     y = y+1;
    }
    x  = xD;
    y  = yD;
```

```
    hit = 0;

    while(x<=xB)
    {
     for(y=yD;y<=yB;y++)
     {
printf("x= %d y= %d\n",x,y);
      if(((DAT[x][y]&WRITE_MASK)>>6)==1)
      {
       hit = hit+1;

       shade       =  DAT[x][y]&GSV_MASK;
       ucini       = (DAT[x][y]&UCI_MASK);
       heighti     = (DAT[x][y]&VGH_MASK);
       elevi       = (DAT[x][y]&ELEV_MASK);
       surfi       = (DAT[x][y]&SURF_MASK);
       elevbuild   = ((elevi>>20)+(2*bheight))<<20;
       DAT[x][y]   = shade+elevbuild+ucini+heighti+surfi+man+script;

       if(y<=ylow)
        ylow  = y;
       else
        yhigh = y;
      }
     }

    if(hit>1)
    {
     shade       =  DAT[x][ylow]&GSV_MASK;
     ucini       = (DAT[x][ylow]&UCI_MASK);
     heighti     = (DAT[x][ylow]&VGH_MASK);
     surfi       = (DAT[x][ylow]&SURF_MASK);
     elevi       = (DAT[x][ylow]&ELEV_MASK);
     DAT[x][ylow] = shade+elevi+ucini+heighti+man+script+surfi+south;

     shade       =  DAT[x][yhigh]&GSV_MASK;
     ucini       = (DAT[x][yhigh]&UCI_MASK);
     heighti     = (DAT[x][yhigh]&VGH_MASK);
     surfi       = (DAT[x][yhigh]&SURF_MASK);
     elevi       = (DAT[x][yhigh]&ELEV_MASK);
     DAT[x][yhigh] = shade+elevi+ucini+heighti+man+script+surfi+north;
    }
```

```
      if(hit=1)
      {
       shade       =  DAT[x][ylow]&GSV_MASK;
       ucini       = (DAT[x][ylow]&UCI_MASK);
       heighti     = (DAT[x][ylow]&VGH_MASK);
       surfi       = (DAT[x][ylow]&SURF_MASK);
       elevi       = (DAT[x][ylow]&ELEV_MASK);
       DAT[x][ylow] = shade+elevi+ucini+heighti+man+script+surfi+south;
      }

      x = x+1;
     }
    }
   }
   break;

   case ESCKEY:
   if (val == 0)
    run = FALSE;
   break;
  }
 }
}
```

APPENDIX E. TREES.C

```
/*
  PROGRAM:  trees.c
  Tree stand generation algorithms.

  15OCT96
*/

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include"PVG_DEC.IN"

int elev, height, ucin, nature, written;

void TREES(int elevi, float vaveh, float sigmah, int vaveu, int sigmau)
 {
  float randmax, ri, rii, maxvalh, minvalh, testval, maxvalu, minvalu,
       high, testii;

  int testvali, itestii;

  randmax  = RAND_MAX*1.0;
randomh:
  ri       = rand()/randmax;
  maxvalh  = vaveh + sigmah;
  minvalh  = vaveh - sigmah;
  testval  = ri*maxvalh;

  if(testval>minvalh && testval<maxvalh)
  {

   testvali = (int)testval;
   high     = testval;
   height   = testvali<<10;
   elev     = (elevi + (testvali*2))<<20;

randomu:
   rii      = rand()/randmax;
```

```
 maxvalu   = vaveu+(sigmau*1.0);
 minvalu   = vaveu-(sigmau*1.0);
 testii    = rii*maxvalu;
 itestii   = (int)testii;
 if((testii>minvalu && testii<maxvalu && testii<high) || (testii<1.0))
 {
  ucin     = itestii<<18;
  nature   = 1<<7;
  written  = 0<<6;
 }
 else
  goto randomu;
}
else
 goto randomh;
}
```

LIST OF REFERENCES

1. Driels, M. and Lind, J., *Prototype Line of Sight and Target Acquisition Software for High Resolution Databases*, Technical Report, Department of Mechanical Engineering, Naval Postgraduate School, Monterey, California, December 1995.

2. Whitney, M.R., *Visualization of Improved Target Aquisition Algorithm for JANUS(A),* pp. 8-9, Masters Thesis, Department of Mechanical Engineering, Naval Postgraduate School, Monterey, California, December 1994.

3. *Graphics Library Programming Guide*, pp. 5.1-9, Silicon Graphics Computer Systems, 1990.

BIBLIOGRAPHY

Aitken, P. and Jones, B., *Teach Yourself C in 21 Days*, SAMS Publishing, 1994.

Young, J.M., *Synthetic Environments for C3 Operations*, Masters Thesis, Department of Mechanical Engineering, Naval Postgraduate School, Monterey, California, September 1994.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center 2
   8725 John J. Kingman Rd., STE 0944
   Ft. Belvoir, Virginia 22060-6218

2. Dudley Knox Library 2
   Naval Postgraduate School
   411 Dyer Rd.
   Monterey, California 93943-5101

3. Prof. Morris Driels 3
   Attn: Mechanical Engineering Dept., Code ME
   Naval Postgraduate School
   Monterey, California 93943-5000

4. LT Joseph R. Darlak 2
   c/o Mr. James Volk
   4384 Homestead Lane
   Clarence, New York 14031

5. Mr. Bob Bennett 1
   ATRC-WB-WSMR
   New Mexico 88002

6. Mr. Ferney Payan 1
   ATRC-WB-WSMR
   New Mexico 88002

7. Mr. Dave Dixon 1
   ATRC-WB-WSMR
   New Mexico 88002

8. Mr. Dale Robison 1
   NAWC WPNS
   Code 455530d
   China Lake, California 93595

9. Ms. Judy Lind 1
1010 Benito Avenue
Pacific Grove, California 93950

10. MAJ Bill Murphy 1
TRADAOC-MTRY
P.O. Box 8692
Monterey, California 95943-0692

11. LTCOL Ralph Wood 1
Director
TRADAOC-MTRY
P.O. Box 8692
Monterey, California 95943-0692

12. Mr. John Mazz 1
USAMSAA
Attn: AMXSP-CA
Aberdeen Proving Ground, Maryland 21005-5071

13. Dr. Barbara O'Kane 1
NVESD
10221 Burbeck Road
AMSEL-Rd-NV-V
Fort Belvoir, Virginia 22060

14. Dr. W. Baer 1
Attn: Computer Science Dept., Code CS
Naval Postgraduate School
Monterey, California 93943-5000

15. Dr. L. Obert 2
NVESD-V MSB
AMSEL-RD-NV-V
Fort Belvoir, Virginia 22060

16. Dr. T. Doll 1
GTRI
Georgia Tech.
Atlanta, Georgia 30332-0841